UNIVERSITY OF CALIFORNIA
RIVERSIDE

Efficient Algorithms for Identification and Analysis of Repetitive Patterns in Biological Sequences

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Jie Zheng

September 2006

Dissertation Committee:

Dr. Stefano Lonardi, Co-Chairperson
Dr. Tao Jiang, Co-Chairperson
Dr. Timothy J. Close

The Dissertation of Jie Zheng is approved:

Committee Co-Chairperson

Committee Co-Chairperson

University of California, Riverside

Acknowledgments

I am most grateful to my advisor Professor Stefano Lonardi and my co-advisor Professor Tao Jiang, for their guidance in the early and crucial stage of my research career. It is Professor Tao Jiang who introduced me to the exciting fields of computational molecular biology and bioinformatics, and gave me the first research problem to work on. His warm-hearted encouragement has inspired in my heart a love for original research. A dedicated, caring, and stimulating advisor, Professor Stefano Lonardi has provided me with helpful and insightful advice, not limited to technical questions, throughout my Ph.D. studies. He is one of the most patient listeners I have ever met.

I am also grateful to Professor Timothy J. Close (Department of Botany and Plant Sciences) for his support and supervision in the NSF Barley Genome Project. I learned from him how to work fruitfully with biologists.

I would like to thank three post-doctoral researchers at UCR. Dr. Jan T. Svensson provided valuable comments on my work in the OLIGOSPAWN project; Dr. Xin Chen introduced me to the area of genome rearrangement and collaborated with me in the project of ortholog assignment; Dr. Petr Kolman collaborated with me on the MCSP problem.

I wish to express my gratitude to Professor Neal E. Young for teaching me how to design approximation algorithms as well as for helpful discussions on the MCSP problem, Professor Lawrence Harper (Department of Mathematics) for teaching me combinatorics, and Professor Xiao-Song Lin (Department of Mathematics) for discussions on applying topology to computational biology.

My thanks also go to my friends and UCR Alumni, Andres Figueroa, Li Jia, Jing Li, Yu Luo, Jianjun Tian, Chuhu Yang for their friendship.

Finally, I would like to thank my parents and my wife Yuan for their support and love.

The text of this dissertation is in part rewritten from the following previously published material.

- J. Zheng, S. Lonardi. Discovery of repetitive patterns in DNA with accurate boundaries. Proc. of IEEE International Symposium on BioInformatics and BioEngineering (BIBE'05), pp. 105-112, Minneapolis, Minnesota, USA, 2005.

  Stefano Lonardi supervised the research.

- J. Zheng, T. Close, T. Jiang, S. Lonardi. Efficient Selection of Unique and Popular Oligos for Large EST Databases. Proc. of Symposium on Combinatorial Pattern Matching (CPM'03), pp. 384-401, LNCS 2676, Morelia, Mexico, 2003.

  J. Zheng, T. Close, T. Jiang, S. Lonardi. Efficient Selection of Unique and Popular Oligos for Large EST Databases. Bioinformatics, vol. 20, no. 13, pp. 2101-2112, 2004.

  J. Zheng, J. Svensson, K. Madishetty, T. Close, T. Jiang, S. Lonardi. OligoSpawn: a software tool for the design of overgo probes from large unigene databases. BMC Bioinformatics, 7:7, 2006.

  Timothy J. Close, Tao Jiang, and Stefano Lonardi directed and supervised the research, and Jan T. Svensson and Kavitha Madishetty contributed to the publication.

- A. Goldstein, P. Kolman, J. Zheng. Minimum common string partition problem: hardness and approximation. Proc. of International Symposium on Algorithms and Computation (ISAAC'04), pp. 484-495, LNCS 3341, Hong Kong, China, 2004.

  A. Goldstein, P. Kolman, J. Zheng. Minimum common string partition problem: hardness and approximation. Electronic Journal of Combinatorics, vol. 12(1), 2005.

  Avraham Goldstein and Petr Kolman contributed to the publication through discussions.

The above previously published material has been incorporated in this dissertation with kind permission of Springer Science and Business Media, Oxford University Press, and IEEE Computer Society.

ABSTRACT OF THE DISSERTATION

Efficient Algorithms for Identification and Analysis of Repetitive Patterns in Biological Sequences

by

Jie Zheng

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2006
Dr. Stefano Lonardi, Dr. Tao Jiang, Co-Chairpersons

Repetitive patterns, or ”repeats” for short, are sequences that occurs repeatedly in biological sequences. Despite their conceptual simplicity, the problem of discovering and characterizing biologically significant repeats is still open. There are three main stages of repeat analysis: definition, identification, and interpretation. In this dissertation, we tackle the computational challenges in each stage and demonstrate the utility of our approaches to several fields of computational biology.

As it turns out, devising a definition of repeats that is mathematically sound and biologically plausible is rather challenging. We propose a new definition of repeats that considers both length and frequency of occurrences. Our approach is a two-step process: we first model the building blocks of repeats (called *elementary* repeats), and then model the longer repeats as concatenations of elementary repeats (called *composite* repeats).

In the identification stage, the challenge is to design high-throughput algorithms to process large datasets. We describe algorithms for selecting two types of oligos from large DNA sequence databases, namely (i) *unique* oligos, each of which appears (exactly) in one sequence but does not appear (exactly or approximately) in any other sequence, and (ii) *popular* oligos, which appear (exactly or approximately) in many sequences. By taking into account the distribution of the short substrings in the sequence database, the algorithms show remarkable efficiency. We implemented our algorithms in a software called OLIGOSPAWN, which has been used successfully to process a large unigene database for barley.

The goal of the interpretation stage is to infer evolutionary or functional relations among the discovered repeats. In the wide range of problems in this domain, we address the problem of genome rearrangement with multigene families, where duplicated genes are treated as repetitive patterns. We aim to find a one-to-one mapping of genes between two genomes such that their breakpoint distance is minimized, which is formulated as minimum common string partition (MCSP) problem. We prove MCSP NP-hard and give approximation algorithms.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

> *Mathematics is the art of giving the same name to different things. (As opposed to the quotation: Poetry is the art of giving different names to the same thing.)*
>
> Jules Henri Poincaré (1854-1912)

Large-scale genome, transcriptome and proteome projects, especially the Human Genome Project, have produced and continue to produce tremendous amounts of data. These data can be roughly classified by their intrinsic dimensionality. One-dimensional data include biological sequences, e.g. DNA, RNA, and protein sequences. Two-dimensional data include gene expression data, RNA secondary structures, and mass spectrometry data, among others. Three-dimensional data include the tertiary structures of proteins as a notable example. This dissertation is concerned with the analysis of one-dimensional data.

Biological macromolecules (i.e. DNA, RNA, and proteins) consist of chains of small organic molecules called *monomers*. Despite the overall complexity of macromolecules,

they are composed by only a few different types of monomers. In DNA, the monomers are adenine, guanine, cytosine, and thymine. RNA uses the same set of monomers with the exception of thymine which is replaced by uracil. In proteins, there are 20 types of monomers called *amino acids*. From a combinatorial viewpoint, these are discrete "objects" and can be represented as sequences (or "strings") over finite alphabets consisting of symbols representing the monomers. This natural representation as strings enables us to take advantage of the large corpus of computer science research on text processing, in particular regarding algorithms and data structures. Developing efficient computer algorithms for the analysis of biological sequence data is arguably the main objective of a subfield of computer science called *computational molecular biology*.

In biological sequence analysis, the discovery and characterization of repetitive patterns (or "repeats" for short) plays a central role. Roughly speaking, repeats are strings that appear repeatedly. It is assumed that frequent or rare patterns may correspond to signals in biological sequences that imply functional or evolutionary relationships. For example, repetitive patterns in the promoter regions of functionally related genes may correspond to regulatory elements (e.g. transcription factor binding sites). As another example, a multigene family is a group of genes from the same organism that encode proteins with similar sequences. It is believed that a multigene family is formed through DNA duplications and variations of a single ancestral gene.

As it turns out, the concept of repeat is one of the most versatile in computational biology. An example can be found in [48], where the authors applied repeat analysis to five distinct

areas of computational biology: checking fragment assemblies, searching for low copy repeats related to human malformations, finding unique sequences, comparing gene structures and mapping of cDNA/EST. In this dissertation, we aim to build new models of repeats that are more realistic and flexible than in [48], and to apply repeat analysis to several application domains.

1.1 Challenges

Based on our experience, there are three main stages of repeat analysis: definition, identification, and interpretation. Next, we describe the challenges in each stage.

Definition. The majority of existing methods (e.g. [48, 34]) define a repeat as a *pair* of similar strings of maximal length. They tend to ignore, however, the importance of repeat *frequency*, i.e. the number of occurrences. It turns out that finding a definition of repeat that takes into account both length and frequency is rather challenging. Also, the definition is supposed to incorporate the notion of similarity between occurrences of the same repeat, which adds to the difficulty.

Identification. The challenge in this stage is to design algorithms with high-throughput efficiency. Typical input consists of large-scale datasets (e.g. whole-genome sequences). Since the consensus sequence of repetitive patterns may not appear exactly in the input at all, the search space is in general much bigger than the input itself. Moreover, space efficiency

is often crucial in sequence analysis, because the limitation of computer memory can make it difficult to process large input no matter how fast the algorithms can be. Therefore, our algorithms should achieve good trade-offs between time and space efficiency.

Interpretation. In this stage, the goal is to analyze the discovered repeats in order to infer their functional and evolutionary relationships. To this end, we need to visualize the repeats comprehensively and intuitively. The challenges are mainly due to the overwhelming number of repeats to handle. Moreover, we need to rank different repeats by either statistical or biological significance. However, there are different ways to assess statistical significance, and statistical significance may not correspond to biological significance.

1.2 A summary of our contribution

In this dissertation, we tackle some of the challenges mentioned above and present algorithms for finding and analyzing repeats.

In Chapter 2, we give a novel definition of repeats that takes into account both length and frequency. The main idea is to employ a two-level approach: first, we identify short and frequent repeats as basic building blocks, which are called *elementary* repeats; second, we identify long and less frequent repeats as concatenations of elementary repeats, which are called *composite* repeats. We also design efficient algorithms for finding elementary repeats and report simulation results on synthetic data.

In Chapter 3, we present efficient algorithms for selecting short (e.g. 20-50 bases) strings

called *oligos* from large EST (*Expressed sequence tags*) unigene databases. ESTs are partial sequences of expressed genes generated by sequencing from one or both ends of cDNA sequences, and unigenes are assemblies of overlapping EST sequences or non-overlapping EST sequences (see Section 3.1 in Chapter 3 for more details). We design two complementary types of oligos: (i) unique oligos, each of which occurs exactly in one unigene, but does not occur exactly or approximately in any other unigene. (ii) popular oligos, which appear exactly or approximately in many unigenes. The main strategy is based on the analysis of the frequencies of short substrings in the unigene database. The resulting algorithms show remarkable efficiency. We have implemented the algorithms into a software tool called OLIGOSPAWN, which has been used successfully to process a large unigene database for barley.

In Chapter 4, we address the problem of genome rearrangement with multigene families, where duplicated genes are treated as repetitive patterns. In particular, we aim to find a one-to-one mapping of genes between two genomes such that their breakpoint distance (see e.g. [75]) is minimized, which is formulated as minimum common string partition (MCSP) problem. The correspondence of genes implies evolutionary and functional relations of homologous genes between two species. We prove that MCSP is NP-hard even when each gene family contains no more than two genes, and we present several approximation algorithms with ratio bounds.

The main contribution of Chapter 2 is the novel definition of repeat. Chapter 3 is focused on the efficiency of algorithms. Chapter 4 aims to find evolutionary relationships among

discovered repeats. Thus, each of the three chapters focuses on one of the three stages of repeat analysis. However, we do not claim to have solved all the problems in each stage. Rather, we present promising techniques for solving specific problems arising from concrete applications.

Each of the three chapters is organized as follows. At the beginning, we describe the biological motivation and related work. Then, we introduce definitions and computational problems, and study their properties (e.g. computational complexity). After that we present algorithms for solving the problems and analyze their theoretical performance (e.g., asymptotic running time or approximation ratio bounds). Finally, we perform empirical analysis of the algorithms by simulation on synthetic or real biological data.

1.3 Targeted audience

In addition to computational biologists, two other types of readers may find the material in this dissertation useful. First, this dissertation could provide biologists with several practical tools for discovering and analyzing patterns from large sequence databases. Second, computer scientists whose research interest is in information retrieval, data mining and data compression, etc., may possibly extend some of the ideas presented here to their own fields of study, due to the generality of the concept of repetitive patterns.

It is appropriate to mention topics that are related but not treated in this dissertation. First, we do not attempt to analyze biological data directly. Instead, we provide useful computa-

tional tools that facilitate the biological discovery. Our experiments mainly test the accuracy and efficiency of the algorithms. Second, this dissertation is application-oriented. Our objective is to design and implement practical algorithms, instead of exploring theoretical properties of the computational problems. Last but not least, our methods are combinatorial, and do not employ statistical models (e.g. [74, 23, 27]) or machine learning techniques (e.g. [5]).

Chapter 2

Repeat Finding

In this chapter we give a novel definition of repeats that balances the importance of the length and the frequency of repeats. We model the basic building blocks of repeats called elementary repeats, which leads to a natural definition of repeat boundaries. We also design efficient algorithms for finding elementary repeats, and test them on synthetic data. This chapter is a revision of [79].

2.1 Preliminaries

Despite the incredible progresses that have occurred in Genomics in the last fifty years, several fundamental questions remain unanswered. Some of the most intriguing questions are about the role of non-coding DNA, and in particular the role of repetitive sequences. As it turns out, a significant fraction of the genome of complex organisms is repetitive. For example, only about 1% of the human genome codes for proteins, but more than 50% of

the human genome is composed of repetitive sequences [51]. Repeats are classified in five classes, namely pseudo-genes (0.1% of the human genome), simple repeats (3%), segmental duplications (5%), transposons (45%) and tandem repeats.

Although these long stretches of repeated DNA are commonly regarded as "junk", there is evidence that a variety of genetic diseases are associated with defects in the repeated structure of some regions of the human genome. More than a dozen human genetic diseases, including fragile X syndrome, myotonic dystrophy and Friedreich's ataxia are related to irregularities in the length of repeats. Insertional mutagenesis by SINEs (short interspersed elements) and LINEs (long interspersed elements) in mammals have resulted in cases of hemophilia, Duchenne muscular dystrophy, and sporadic breast and colon cancer.

A very important problem in computational biology is how to identify, classify, represent and visualize repeats. The general problem of repeat discovery and classification can be decomposed into two distinct subproblems. In the first, one identifies the boundaries of each copy of the repeats. This problem, usually attacked by performing local alignments, is difficult because *of degraded or partially deleted copies, related by distinct repeats, and segmental duplication covering more than one repeat* (quote from [6]). The second problem in the repeat classification is to infer the series of duplications that produced the final string. In this paper, we will focus on the first problem.

2.1.1 Related work

Most of the computational tools for repeat identification available to biologists either require an annotated library of repeats (e.g., REPEATMASKER [70]) or simply output all pairs of repeated regions (e.g., REPUTER [49, 48], [68]). The problem with the first approach is that it is too limiting, in particular when trying to analyze a new genome for which a complete repeat library is unavailable. The latter approach is not completely satisfactory either, because it fails to elucidate the complex structure of repeats.

From a theoretical point of view, a repeat is usually defined as a pair of identical (or similar) substrings. For example, in Gusfield's definition [34] the objective is to find maximal repeated pairs. In REPUTER [49, 48] the output is again a list of pairs of similar strings of maximal length. Most definitions have in common the idea of maximizing the *length* of the repeats. They tend to ignore, however, the importance of repeat frequency, i.e., the number of occurrences.

As said above, repetitive elements typically occur more than twice in real biological sequences. For example, transposons typically occur hundreds of thousands of times in complex genomes. Therefore, we believe that a biologically meaningful definition of repeats must take into account both the length *and* the frequency of repeats (and perhaps other factors as well). Unfortunately, as noted by some researchers (e.g., [8]), devising a definition of repeats which is biologically plausible is not an easy task. When the frequency of repeats are allowed to be more than twice, there exist tools (e.g., [66, 9]) for finding short or tandem repeats. However, they are unable to find long and dispersed repeats.

Bao and Eddy [6] and Pevzner *et. al.* [59] recognized the shortcomings of definitions that rely solely on length, and attempted to take into account other factors. A rigorous definition of repeats considering both length and frequency, however, has not been given yet. The difficulty lies in the assessment of repeat boundaries once the objective of maximality in length is dropped. According to [6], approaches by multiple alignment are also problematic because the mosaic structure of repeats will be missed and the problem of multiple alignment itself is difficult. In [59], Pevzner *et. al.* proposed a graph called *A-Bruijn graph* to explore the mosaic structure of repeats. However, A-Bruijn graph is complicated and difficult to analyze, especially when the input sequences are long and contain a large number of repeats. In addition, the approach based on A-Bruijn graph still requires the pairwise local alignment of the input sequences given as input, and thus the results heavily depend on the performance of the pairwise alignment.

Recently, more attention has focused on the detection of repeat boundaries. Price *et. al.* [60] proposed the tool REPEATSCOUT to identify repeat boundaries via extension of consensus seeds. R. Edgar and E. Myers [24] developed the tool PILER to find repeats with reliable boundaries by considering well-known repetitive structures, e.g., terminal repeats and tandem arrays.

2.1.2 Our contribution

As noted by Bao and Eddy, *the problem of automated repeat sequence family classification is inherently messy and ill-defined and does not appear to be amenable to a clean algorithmic*

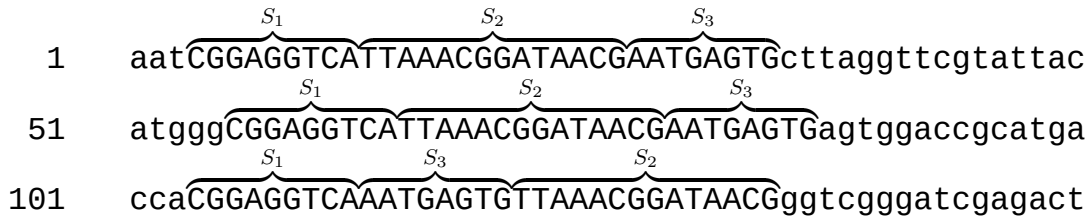

Figure 2.1: An example of elementary repeats.

attack (quote from [6]). We felt that in order to give a clean algorithmic attack, a natural and clean definition of repeats was indispensable. To this end, we propose a bottom-up approach that starts with the definition of basic building blocks of repeats that we call *elementary repeats*. Intuitively, an elementary repeat is a string whose substrings have similar (or identical) distribution of occurrences. Before giving the formal definition, let us consider an example in Figure 2.1 which illustrates the motivations as well as the basic idea.

In Figure 2.1, the DNA sequence contains three repeats S_1, S_2, and S_3, each occurring three times. At the first glance, it would appear that the segment $A = S_1S_2S_3$ is the "correct" repeat, as it is the longest. This is, in fact, what would be reported if the definition were solely based on maximal length. However, we can see that A is actually composed of independent "elementary" repeats, namely S_1, S_2, and S_3, because their order is shuffled in the third occurrence.

The example in Figure 2.1 is not a fictional scenario. Instead, it corresponds to a real biological case. In the four groups of LTR retrotransposons, Ty1/copia differs from Ty3/gypsy, Bel elements, and retroviruses by the order of POL protein coding domains. The order in Ty1/copia elements is *protease, integrase, RT/ribonuclease H*; while the order in the other

three groups is *protease, RT/ribonuclease H, integrase*. The shuffling of the POL domains is one of the main features in the classification of LTR retrotransposons (see [29] for more details).

The identification of the internal components of complex repeats, such as POL domains of LTR retrotransposons, must be the first step in the accurate detection of repeats. The discovery of the internal structure could be helpful also to infer the functions of repeats because the order of these basic blocks is sometimes responsible for their functions. Moreover, conservation of the basic blocks among different types of repeats can reveal their evolutionary relationship.

The aim of this chapter is to model the basic building blocks of repeats as combinatorial objects. To this end, we have defined a specific set of properties that they must satisfy. First, the blocks must occur a minimum number of times and can not be too short. Second, they must be *elementary*, i.e., they can not contain other basic blocks inside. We call these basic blocks *elementary* repeats. In this paper, we consider two types of elementary repeats, *exact* elementary repeats and *approximate* elementary repeats. The former type requires all copies of one repeat to be exactly identical, while the latter allows substitutions, insertions and deletions.

2.2 Definition

We use standard concepts and notation about strings. The set Σ denotes a nonempty *alphabet* of *symbols*, and a *string* over Σ is an ordered sequence of symbols from the alphabet. We assume that the input is a string S of length n over Σ. We also use $|S|$ to denote the length of S. We write $S[i]$, $1 \leq i \leq n$, to indicate the i-th symbol in S. We use $S[i, j]$ as shorthand for the substring $S[i]S[i + 1] \dots S[j]$ where $1 \leq i \leq j \leq n$, with the convention that $S[i, i] = S[i]$.

Let A be a substring that occurs (exactly) multiple times in the input string S. Let $(A_1, A_2, \dots, A_f)$ be the sorted list of the occurrences of A. We call f the *frequency* of A, also denoted $f(A)$. In the rest of the paper, let $f_m \geq 2$ denote the minimum frequency of repeats, which is a parameter set according to applications. Although the symbol A_i denotes the sequence composition of one of the occurrences of A, sometimes we will abuse this notation and used it to denote the position of A in S. The meaning will be clear from the context. We call A_i a *copy* of A in S.

Let B be a substring of A, and let $(B_1, B_2, \dots, B_k)$ be the sorted list of the copies of B in S. Clearly, each copy of B is a substring of a copy of A, but B may also appear elsewhere in S. We say that B occurs with *shift* s in A if B starts at position $s + 1$ in A, that is $A[s + 1, s + |B|] = B$.

Definition 2.2.1 *Let A and B be substrings of S as defined above. Then B is a* subrepeat *of A if $f = k$ and every B_i occurs with the same shift s in A_i for all $i = 1, 2, \dots, m$.*

Intuitively, B is a subrepeat of A when the distribution of the occurrences of B in S "agrees" with that of A. Next, we need to consider the length of repeats. A substring of S is called *nontrivial* if its length is at least equal to a specified threshold $l_{\min}$. Hereafter, we will always use $l_{\min}$ to denote such threshold. Typically, $l_{\min} \approx \log_{|\Sigma|} n$, but it can be set as other values according to the application.

Definition 2.2.2 *An* exact elementary repeat *of S is a nontrivial substring A of maximal length such that A occurs at least f_m times and every nontrivial substring of A is a subrepeat of A.*

In other words, A is an exact elementary repeat when it does not contain any nontrivial substring with a different distribution of occurrences and it is maximal in length. Going back to the example in Figure 2.1, if $l_{\min} = 8$, then S_1, S_2, and S_3 are exact elementary repeats because they do not contain any substring of length at least 8 with a different distribution of occurrences. However, the concatenation $A = S_1S_2S_3$ of the three substrings is not an exact elementary repeat, because nontrivial substrings S_1, S_2, and S_3 are not subrepeats of A.

Now let us generalize exact elementary repeats by considering the approximate case. Given a real number $\epsilon \geq 0$, we say that sequence A *ϵ-matches* sequence A' if $D(A, A')/|A| \leq \epsilon$, where $D(A, A')$ is the *edit distance* between A and A', i.e. the minimum number of edit operations (substitutions, insertions and deletions) that transform A into A'. We call A' as an *ϵ-copy* of A. In addition, let B be a substring of A. Then there must be a substring in A', denoted by B', which corresponds to B. We call B' the *image* of B. The concept of "image" mirrors the concept of "occurrence with shift s" in the exact case. Thus we will define it

formally in the following. Given an edit script $\mathcal{E} = (e_1, e_2, \ldots, e_d)$ that transforms A into A', we say that $\mathcal{E}$ is *ordered* if, for any i, j such that e_i, e_j operate on $A[i']$, $A[j']$ respectively, $i < j$ implies $i' < j'$. That is, $\mathcal{E}$ operates on A from left to right.

Definition 2.2.3 *Suppose the shortest ordered edit script that transforms A into A' is $(e_1, e_2, \ldots, e_t)$, where $(e_i, e_{i+1}, \ldots, e_j)$, for $1 \leq i < j \leq t$, transforms a substring B of A into a substring B' of A'. Then, we call B' as the* image of B induced by A.

String B is called an *internal repeat* of A if (i) B is a substring of A, (ii) every image of B induced by A is an ϵ-copy of B, and (iii) the total number of non-overlapping ϵ-copies of B in S is equal to the images of B induced by A. Note that internal repeats should not be reported because they are not independent repeats.

Definition 2.2.4 *An* approximate elementary repeat *of S is a nontrivial string A of maximal length such that A has at least f_m ϵ-copies in S and every nontrivial substring of A is an internal repeat of A.*

In contrast to exact case, an approximate elementary repeat need not appear exactly in S. As a result, approximate repeats are much harder to identify than exact repeats. In the rest of the section, we may sometimes omit the word *elementary* for brevity.

Lemma 2.2.1 *The number of copies of* all *exact repeats in S is upper bounded by n.*

In fact, for any two exact repeats A and B, A is not a substring of B; otherwise, A can be extended to B and remains an exact repeat, contradicting the maximality of length in

Definition 2.2.2. This is important because the size of output is a serious practical issue in repeat analysis.

2.3 Algorithms for finding elementary repeats

The goal in this section is to design algorithms for finding both exact and approximate repeats in a long input sequence. A repeat can be represented by a list of pairs, each denoting the left and the right boundaries of a repeat copy. Hence, the problem of identifying repeats can be restated as the problem of finding their accurate boundaries. The task of our algorithms is to scan the input string S and decide whether a position of S is a repeat boundary or not.

2.3.1 Exact repeats

An exhaustive algorithm for finding exact repeats is computationally impractical, because there are $\Theta(n^2)$ substrings in S, and each substring of length l contains $\Theta(l^2)$ substrings to check.

In sequence analysis the q-*gram* approach is well-known for its effectiveness in filtering out non-candidates (e.g., see [34] and reference therein). The idea is to collect the occurrences (or other statistics) for all the substrings of a given length q and use that information to discard substrings that can not be solutions to the problem at hand. Recently, the q-gram approach has been used in repeat analysis (e.g., [73, 24]).

Our definition of exact repeat allows the q-gram filtering approach for the identification

of potential elementary repeats. More specifically, we collect statistics of q-mers in S. We call these q-mers *seeds*.

We find the boundaries of exact repeats by detecting the positions in which there is a change in the distribution of the occurrences of the seeds. Given the definitions in Section 2.2, a necessary condition for a substring A to be an exact repeat is that all its nontrivial substrings are equally frequent. This condition allows us to reduce the search space of potential candidates considerably. Moreover, it suffices to consider only the frequencies of seeds, due to the following lemma.

Lemma 2.3.1 *A nontrivial substring A which occurs at least f_m times is an exact elementary repeat if and only if it is a maximal substring such that all its substrings of length $l_{\min}$ are as frequent as A itself.*

Proof. If A is an exact repeat, then by definition all its nontrivial substrings, including those of length $l_{\min}$, are as frequent as itself. If A is not an exact repeat, either it is not maximal in length, or it has a nontrivial substring B that is more frequent than A. But then any $l_{\min}$-mer in B, which is also in A, is more frequent than A. □

Our goal is to compare the frequency of the substring A with its seeds efficiently. Our strategy is to store and retrieve the frequencies of substrings in the suffix-tree T for string S. In the following, the notations about suffix-tree are adopted from [34, pages 90–91].

Lemma 2.3.2 ([34]) *In linear time, we can compute for each internal node v the number of leaves in v's subtree.*

Proof. We can do a depth-first search on T in linear time and compute the number of leaves in v's subtree by adding the numbers of leaves of v's children. □

Let f_i denote the frequency of the ith q-mer seed, and let $f(A)$ denote the frequency of any substring A in S. Every substring A of S can be associated with a node v in T, such that $f(A)$ equals the number of leaves in v's subtree. To find such a node v, we match the symbols of A along the unique path in T until A is exhausted. If the path ends at a node, then that node is v; if the path ends in the middle of an edge, then the lower end of the edge is the node v associated with A. Hence, there is a many-to-one mapping from substrings of S to the nodes of T. Moreover, two substrings of equal length are mapped to the same node if and only if they are identical.

Lemma 2.3.3 *In linear time, we can map every q-mer seed of string S to a node v in suffix-tree T with minimum string-depth such that the seed is a prefix of the path-label of v.*

Proof. We build a vector V of length $n - q + 1$ such that $V[i]$ contains the pointer to the suffix-tree node associated with the ith seed. We do a depth-first search on T and record the string-depth (i.e. length of path from root) of the node being visited. If by moving from node u to node v the string-depth increases from smaller than q to at least q, then let P_v be the pointer to node v. For every leaf i in v's subtree, which we visit one by one in the traversal, we fill $V[i]$ with P_v. Since depth-first search takes $O(n)$ steps and each operation above takes constant time, building the vector V takes linear time. □

Corollary 2.3.1 *After linear-time preprocessing, the frequency f_i of the ith seed is equal*

to the number of leaves in the subtree under node pointed to by $V[i]$, and it thus can be computed in constant time.

The algorithm Exact-Repeat for finding exact elementary repeats works as follows.

Phase 1. Construct suffix tree T for string S, and we associate with each internal node v of T the number of leaves in v's subtree. We also construct the vector $V[1 \ldots n-q+1]$ as described in the proof of Lemma 2.3.3.

Phase 2. Scan string S from left to right, and for $i = 1, \ldots, n-q+1$, decide if i is the left or right (or neither) boundary of an exact repeat. The left and right boundaries locate alternatively along S. Initially, we set i to $\min\{k | f_k \geq f_m\}$ as the left boundary. Suppose we have set position b_l as a left boundary, and we look for the right boundary b_r. Let i increase from b_l, and let $A_i = S[b_l, i+q-1]$, i.e. A_i is the substring of S from b_l to the end of the ith seed. If $f(A_i) = f_i$, we move on to $i+1$ since, by Lemma 2.3.1, A_i is a prefix of an exact repeat; otherwise, we set b_r to $i-1$ as the right boundary paired with b_l. Then, we move on to find next left boundary $j = \min\{k | k > b_r, f_k \geq f_m\}$, until string S is exhausted.

To efficiently compute $f(A_i)$, we keep track of the last symbol of string A_i along the path in suffix tree T as follows. If i is a new left boundary, we jump to node of T pointed to by $V[i]$ and locate the end of the path whose label is identical with the ith seed. When moving from i to $i+1$, we match the last symbol of A_{i+1} along the path in T, increasing string-depth by one. By Lemma 2.3.2 and Lemma 2.3.3, in the ith iteration, we can compute f_i and $f(A_i)$ in constant time.

In order to retrieve positions of repeat images in the next phase, whenever a right boundary is selected, we mark the most recently visited node, whose path-label is the identified repeat.

Phase 3. Finally, we do a depth-first traversal of T so that for each marked node whose path-label is a repeat, we collect the leaves in its subtree, which contains positions of the repeat images. The repeat length is the string-depth of the node.

Time and Space Complexity. In phase 1, construction of suffix tree and vector V takes linear time, due to Lemma 2.3.3. Phase 2 takes linear time, since each position i is examined once and the ith iteration takes constant time. In phase 3, depth-first traversal of T takes linear time. Therefore, algorithm Exact-Repeat is in linear time. Similarly, it takes linear space also.

2.3.2 Approximate repeats

Compared with exact case, approximate repeats are more realistic for applications in molecular biology. However, the problem of finding approximate repeats precisely is very difficult, therefore we give a heuristic which extends the ideas for finding exact repeats to follow the definition for approximate repeats.

The basic idea for finding approximate repeats is the following. Suppose a repeat A ϵ-matches $A_1, A_2, \ldots, A_m$ in S, and two seeds of A, say H and J, occur in A with shifts s_H, s_J respectively. Then the images of H induced by A are likely to occur in $A_1, \ldots, A_m$ with

shifts close to s_H. Similarly, images of J are likely to occur in the copies of A with shifts close to s_J. Thus, in each copy of A the offset of the images of H and J is close to $s_H - s_J$. If we subtract the offset, then the occurrence lists of H and J shall be similar, where the similarity is measured by number of images of H that are close to images of J and vice versa. In our algorithm, we use the similarity between the occurrence lists of two *successive* seeds to measure the likelihood that they belong to the same approximate repeat.

For example, $S =$ `cACGTGagACGAGgcaACGTG`, where the capital letters represent repeat `ACGTG` which occurs three times. Suppose the length of seed is two. The occurrences of seed `AC` are 2, 9, and 17, and the occurrences of seed `CG` are 3, 10, and 18. The two seeds have similar occurrence lists except for a shift of one position, hence they are likely to belong to the same repeat. Notice the substitution in the second occurrence `ACGAG`, which is acceptable in our heuristic.

The algorithm Approximate-Repeat is sketched as follows.

Algorithm Approximate-Repeat(S, q, f_m)

Input: string S of length n, seed length q, minimum repeat frequency f_m

Output: left and right boundaries of approximate repeats in S

1. $(B_1, B_2, \ldots, B_k) \leftarrow$ blocks of seeds of frequencies $\geq f_m$
2. **for** each selected block B_i **do**
3. APX-SEED-MERGE (B_i)
4. save contigs from APX-SEED-MERGE into set C
5. CONTIG-MERGE (C)

Subroutine APX-SEED-MERGE ($\mathcal{B}$: a block of seeds of frequencies $\geq f_m$)

6. **for** $i \leftarrow 1$ **to** $|\mathcal{B}| - 1$ **do**

7. $(P, Q) \leftarrow$ sorted lists of occurrences of seed $(i, i+1)$

8. **if** SIM-SCORE $(P, Q, 1) \geq SS_{\min}(P, Q)$ **or**

9. SIM-SCORE $(Q, P, -1) \geq SS_{\min}(Q, P)$ **then**

10. merge seeds i, $i+1$

Subroutine SIM-SCORE (occurrence lists P and Q, offset d)

11. $Q \leftarrow Q - d$

12. $b, p \leftarrow 0$ /* bonus b and penalty p */

13. **for** $i \leftarrow 1$ **to** $|P|$ **do**

14. $Q_j \leftarrow$ *pal* of P_i in Q /* Q_j is the closest to P_i in Q*/

15. **if** $|P_i - Q_j| \leq d_m$ **then** $b \leftarrow b + 1$

16. **else** $p \leftarrow p + 1$

17. **return** $b - \ln(p)$

Subroutine CONTIG-MERGE (list of contigs C)

18. **for** $i \leftarrow 1$ **to** $|C| - 1$ **do**

19. $A \leftarrow$ last seed of contig C_i

20. $B \leftarrow$ first seed of contig C_{i+1}

21. $d_{AB} \leftarrow$ position offset of B against A

22. **if** $d_{AB} \leq D_{\min}$ **then** /* A, B are close enough */

23. $(P, Q) \leftarrow$ sorted occurrence lists of seeds (A, B)

24. **if** SIM-SCORE (P, Q, d_{AB}) $\geq SS_{\min}(P, Q)$ **or**

25. SIM-SCORE (Q, P, $-d_{AB}$) $\geq SS_{\min}(Q, P)$ **then**

26. merge contigs C_i, C_{i+1}

The algorithm Approximate-Repeat works as follows. In line 1, we locate the blocks consisting of seeds that are at least as frequent as a specified threshold f_m. This step reduces the search space by discarding non-repetitive regions. The **for** loop of lines 2-4 merges successive seeds in each selected block into longer substrings if their occurrences are similar. We call these longer substrings, which are constructed by merging one or more successive seeds, *contigs*. Similarly, line 5 merges successive contigs if they belong to the same repeat.

In the subroutine APX-SEED-MERGE, we decide whether a pair of successive seeds should be merged based on the similarity scores of their occurrence lists. Note that the scores and the thresholds are related to the order of seeds. The subroutine SIM-SCORE calculates the similarity score, by first removing the offset in line 11. Then in the **for** loop of lines 13-16, we count the number of positions in P that have a close position in Q. The *pal* of P_i is the position Q_j in Q with the smallest absolute difference with P_i. If the difference is no more than a specified threshold d_m, then we increase bonus b by one; otherwise, we increase penalty p by one. The similarity score is defined as $b - \ln p$, a heuristic formula that emphasizes the significance of matches. The heuristic threshold $SS_{\min}(P, Q)$ is defined as a fraction of $|P|$.

The subroutine CONTIG-MERGE is similar to APX-SEED-MERGE, except that now we

are trying to merge the last seed of the previous contig with the first seed of the next contig. The purpose of CONTIG-MERGE is to reconnect segments that are broken by substitutions and indels.

Time and Space Complexity. It takes $O(n)$ time to find the blocks that consist of seeds of frequencies at least f_m. For each distinct seed, SIM-SCORE is called $O(n)$ times, as in APX-SEED-MERGE and CONTIG-MERGE. Each call of SIM-SCORE takes time linear in the size of input occurrence lists. Therefore, the total running time of algorithm Approximate-Repeat is $O(n^2)$. On average, however, it is much faster, as most seeds are likely to occur only a few times. The algorithm takes again linear space.

Post processing The output contains false positives due to the following reason. Assume the copies of a repeat A are A_1, $A_2, \ldots,$ A_m. Let $\{x_1, x_2, \ldots, x_m\}$ be the multiset of symbols that follow the occurrences of A. When $m > |\Sigma|$, at least $\lfloor m/|\Sigma| \rfloor$ symbols in $\{x_1, x_2, \ldots, x_m\}$ are identical. Let us assume that they are identical with symbol y. Then, the suffix of length $l_{\min} - 1$ of A concatenated with y will result in a spurious repeat at the right end of A. Similarly, a spurious repeat may also occur at the left end of A.

The purpose of post processing is to detect and remove these false positives. Observe that a spurious repeat overlaps with a true repeat with $l_{\min} - 1$ symbols, and it is less frequent than the true repeat. We can detect the spurious repeats by comparing each repeat candidate with its preceding and succeeding repeats. Clearly, the post processing takes time and space linear in n.

2.4 Results

In this section, we demonstrate the accuracy of our algorithms by showing some experimental results. We test our algorithms on synthetic datasets obtained by inserting simulated and real biological repeats into random DNA sequences. We do not compare our results with other tools for repeat identification, because the concept of elementary repeats is unique to this approach while other tools (e.g. REPUTER) output pairs of maximal repeats which are incomparable with our output.

2.4.1 Simulations

Our method of constructing the simulated DNA sequence S is as follows. First, we generate a set of random DNA strings, which corresponds to a set of elementary repeats. Each repeat has attached a specified multiplicity which is generated by Poisson distribution. We randomly permute the multiset of repeat copies by the algorithm of Fisher and Yates [28]. Then, we alternate the repeat copies (selected according to their order in the permutation) with purely random DNA sequences whose lengths are generated by Poisson distribution. The frequencies of repeats and the lengths of the gaps among repeats are selected according to Poisson distribution because it is the most appropriate probabilistic model in this case (see, e.g. [74]).

When we are simulating approximate repeats, we also randomly mutate each copy before it is used. The sequence of edit operations is generated randomly, and the number of muta-

input	output					
gap	S_e (%)			T_p (%)		
ave.	worst	ave.	best	worst	ave.	best
50	92.8	99.9	100	86.3	99.9	100
100	94.7	99.9	100	90	99.8	100
150	100	100	100	100	100	100
200	100	100	100	100	100	100
250	90.4	99.9	100	82.5	99.8	100

Table 2.1: Accuracy of exact elementary repeats detection in simulated data.

tions is bounded by a percentage, say 2%, of the length of the repeat copy. Finally, we append at both ends of the assembled sequence two pieces of random DNA, and that completes the construction of S. We also record the left and the right boundaries of each repeat copy, which will be compared with the output of our algorithms.

When we feed S as input to our algorithms, the output is a list of pairs of boundaries. A boundary in the output matches a boundary in the input if they are within 5 bases of each other. An output repeat copy matches an input one if *both* left and right boundaries match that of the input copy. The *sensitivity*, denoted by S_e, is defined as the ratio of the number of output copies matching the input over the total number of input repeat copies. The *true positive rate*, denoted by T_p, is defined as the ratio of the number of output copies matching the input over the total number of output copies.

In Tables 2.1 we show the accuracy of our algorithm on exact repeats of average length 50. Similarly, Table 2.2 reports the results for approximate repeats of average length 50. In each table, we carried out 5 runs with gap lengths ranging from 50 to 250; for each run, we executed our program 100 times. In every execution, the average number of distinct repeats is

input	output					
gap	S_e (%)			T_p (%)		
ave.	worst	ave.	best	worst	ave.	best
50	95.8	98.8	100	94.6	98.7	100
100	96.6	99.3	100	96.6	99.3	100
150	96.8	99.1	100	96.8	99.0	100
200	94.3	98.0	100	94.3	97.9	100
250	95.1	98.6	100	95.1	98.5	100

Table 2.2: Accuracy of approximate elementary repeats detection in simulated data.

10, the average frequency is 20, and the seed length is 15. Each row of the tables summarizes the results of one run, namely the worst, the average, and the best accuracy of the 100 tests of the run.

From the tables, we can make the following observations. First, the accuracy is remarkably high. The average value of S_e is higher than 98% and the average value of T_p is higher than 95%, except for the approximate repeats of average length 200. Longer approximate repeats are harder to detect, as the edit operations may occur in a narrow interval. In practice, however, elementary repeats are unlikely to be that long. The reason that the accuracy of exact repeat detection is not 100% is that the random DNA strings we generated are not "random" enough. As a result, there are additional repetitive patterns and fragmented repeats, which are not counted in validation but have been captured by our algorithm. Similar situation occurs in approximate case.

Second, the performance of the algorithm for approximate repeats is more consistent than that for exact repeats. This is due to the fact that the algorithm for approximate repeats is more adaptive and thus able to handle more noisy input.

Third, in many cases, especially for approximate repeats, the value of S_e is close to T_p. This happens because often the algorithm detects one correct boundary while the other boundary falls outside 5 bases of the correct one, and it causes a false positive and a false negative simultaneously. These false positives, however, have long overlaps with correct repeats.

2.4.2 Real biological repeats

We also test our algorithms on synthetic data when the repeats are true biological repeats, i.e., we insert copies of real repeats into synthetic DNA sequences. We choose the well-known *Alu* repeats, a family of short interspersed elements (SINEs) that comprises roughly 10% of the human genome. When exact copies of Alu are inserted into random DNA sequences, the average accuracy of our method is above 96%. In approximate case, the average accuracy is above 80%.

Here we test our algorithms on simulated data when the repeats are not random, but true biological repeats. We chose the well-known *Alu* repeats, a family of short interspersed elements (SINEs) that comprises roughly 10% of the human genome. The lengths of real Alu occurrences in human genome range from less than 50 bases to over 400 bases. We fixed the length of the Alu repeat to 281 bases, which is the most typical length.

The accuracy in Table 2.3 is calculated in the same way as that in Table 2.1 and Table 2.2. As shown in Table 2.3, the average accuracy of finding Alu is comparable to that of finding shorter random DNA, as in the Table 2.2. In addition, we found that, while most of the time

input	output			
gap	S_e (%)		T_p (%)	
ave.	exact	approx.	exact	approx.
50	96	84.2	96	82.7
100	99	84.8	99	83.2
150	99	80.6	99	79.0
200	98	79.8	98	78.0
250	97	83.7	97	82.2

Table 2.3: Average accuracy of exact and approximate repeat identification for Alu repeats.

```
GCCTGTAATCCCAGCactttgggaggccgaggtgggcggatcacttgaggtcgggagttc
aagaccagcctggccaacatggcgaaaccccgtctctactaaacataaaaaaattagtca
ggtgtggcggtgccgtGCCTGTAATCCCAGCtattcaggaggctgaggcaccagaattgc
ttgaacccaggaggtggaggttgcagtgaactgaagactgcgccacggcactccagcctg
ggcgacagagcaagactctgtctcaataaataaataattaa
```

Figure 2.2: The sequence of the Alu repeat.

the accuracy is extremely high (typically 100% for exact repeats), sometimes it is surprisingly poor. The reason turned out to be that Alu is not an elementary repeat. Figure 2.2 shows the sequence of the Alu. Note that there are two exact internal repeats of length 15 (shown in upper cases), breaking the Alu repeat into two segments of roughly equal length. It is reasonable to hypothesize that Alu came from two copies of some smaller repeat in the evolutionary history. Using traditional approaches for repeat identification, it is difficult if not impossible to find such kind of "embedded structure" in the repeats. We have also tried our methods on several complete sequences of LTR retrotransposons from TREP database [53], and found similar internal repeats. We suspect that some of them are actually LTRs and other interesting domains.

Chapter 3

Oligo Design

In this chapter we address the problem of designing oligos from large EST unigene databases. We aim to design efficient algorithms for finding two types of oligos, i.e. unique oligos and popular oligos. Observe that finding unique oligos and finding repetitive patterns are somewhat the opposite of each other.

In Section 3.1 we introduce biological motivation, related computational methods, and our contribution. Section 3.2 defines unique oligos and gives an algorithms for finding them. Similarly, Section 3.3 treats popular oligos. In Section 3.4 we describe how to select top candidate oligos according to practical criteria (e.g. GC content, melting temperature, self-annealing). Section 3.5 presents empirical results of using the algorithms on synthetic and real data, and using the designed oligos in wet lab experiments. This chapter is compiled from [77, 78, 80].

3.1 Preliminaries

Expressed sequence tags (ESTs) are partial sequences of expressed genes, usually 200–700 bases long, which are generated by sequencing from one or both ends of cDNA. The information in an EST allows researchers to infer functions of the gene based on similarity to genes of known functions, source of tissue and timing of expression, and genetic map position. EST sequences have become widely accepted as a cost-effective method to gather information about the majority of expressed genes in a number of systems. They can be used to accelerate various research activities, including map-based cloning of genes that control traits, comparative genome analysis, protein identification, and numerous methods that rely on gene-specific oligonucleotides (or *oligos*, for short) such as the DNA microarray technology.

Due to their utility, speed with which they may be obtained, and the low cost associated with this technology, many individual scientists and large genome sequencing centers have been generating hundreds of thousands of ESTs for public use. EST databases have been growing exponentially fast since the first few hundreds sequences obtained in the early nineties by Adams *et al.* [1], and now they represent the largest collection of genetic sequences. As of June 2006, 49 organisms have more than 10^5 ESTs in NCBI's dbEST [11], including barley (*Hordeum vulgare* + *subsp. vulgare*) with 437321 ESTs.

With the advent of whole genome sequencing, it may appear that ESTs have lost some of its appeal. However, the genomes of many organisms that are important to society, including

the majority of crop plants, have not yet been fully sequenced, and the prospects for large-scale funding to support the sequencing of any but a few in the immediate future is slim to none. In addition, several of our most important crop plants have genomes that are of daunting sizes and present special computational challenges because they are composed mostly of highly repetitive DNA. For example, the *Triticeae* (wheat, barley and rye) genomes, each with a size of about 5×10^9 base pairs per haploid genome (this is about twice the size of maize, 12 times the size of rice, and 35 times the size of the Arabidopsis genomes), are too large for us to seriously consider whole genome sequencing at the present time.

Among the set of EST databases, we are especially interested in the dataset of barley (*Hordeum vulgare*). Barley is premiere model for Triticeae plants due to its diploid genome and a rich legacy of mutant collections, germplasm diversity, mapping populations (see `http://www.css.orst.edu/barley/nabgmp/nabgmp.htm`), and the recent accumulation of other genomics resources such as BAC [76] and cDNA libraries [21, 54]. Nearly 300,000 publicly available ESTs derived from barley cDNA libraries are currently present in dbEST. These sequences have been quality-trimmed, cleaned of vector and other contaminating sequences, pre-clustered using the software TGICL (`http://www.tigr.org/tdb/tgi/software/`) and clustered into final assemblies of "contigs" (i.e., overlapping EST sequences) and "singletons" (i.e., non-overlapping EST sequences) using CAP3 [41]. The collection of the singletons and consensus sequences of the contigs, called *unigenes*, form our main dataset. In the rest of the chapter, we consider unigene databases as the input data of our algorithms.

3.1.1 Our Contribution

We study two computational problems arising in the selection of short oligos (*e.g.*, 20–50 bases) from a large unigene database. One is to identify oligos that are *unique* to each unigene in the database. The other is to identify oligos that are popular among the unigenes. More precisely, the *unique oligo* problem asks for the set of all oligos each of which appears (exactly) in one unigene sequence but does not appear (exactly or approximately) in any other unigene sequence, whereas the *popular oligo* problem asks for a list of oligos that appear (exactly or approximately) in the largest number of unigenes. Note that a popular oligo does not necessarily have to appear exactly in any unigene.

A unique oligo can be thought of as a "signature" that distinguishes a unigene from all the others. Unique oligos are particularly valuable as locus-specific PCR primers for placement of unigenes at single positions on a genetic linkage map, on microarrays for studies of the expression of specific genes, and to probe genomic libraries in search of specific genes ([37]).

Popular oligos can be used to screen efficiently large genomic library. They allow one to simultaneously identify a large number of genomic clones that carry expressed genes using a relatively small number of (popular) probes and thus save considerable amounts of money. In particular for the database under analysis, it has been shown previously by a number of independent methods that the expressed genes in Triticeae are concentrated in a small fraction of the total genome. In barley, this portion of the genome, often referred to as the *gene-space*, has been estimated to be only 12% of the total genome [7]. If this is indeed true, then at most 12% of the clones in a typical BAC library would carry expressed genes, and

therefore also the vast majority of barley genes could be sequenced by focusing only on this 12% of the genome. An efficient method to reveal the small portion of BAC clones derived from the gene-space has the potential for tremendous cost savings in the context of obtaining the sequences of the vast majority of barley genes. The most commonly used barley BAC library has a 6.3 fold genome coverage, 17-filter set with a total of 313,344 clones [76]. This number of filters is inconvenient and costly to handle, and the total number of BAC clones is intractable for whole genome physical mapping or sequencing. However, a reduction of this library to a gene-space of only 12% of the total would make it fit onto two filters that would comprise only about 600 Mb. This is about the same size as the rice genome, which has been recently sequenced. A solution for the popular oligo problem should make it possible to develop an effective greedy approach to BAC library screening, enabling a very inexpensive method of identifying a large portion of the BAC clones from the gene-space. This would also likely accelerate progress in many crop plant and other systems that are not being considered for whole genome sequencing.

In this chapter, we present an efficient algorithm to identify all unique oligos in the unigenes and an efficient heuristic algorithm to enumerate the most popular oligos. Although the unique and popular oligos problems are complementary in some sense, the two algorithms are very different because unique oligos are required to appear in the unigenes while the popular oligos are not. In particular, the heuristic algorithm for popular oligos is much more involved than that for unique oligos, although their (average) running times are similar. The algorithms combine well-established algorithmic and data structuring techniques such

as hashing, approximate string matching, and clustering, and take advantage of the facts that (i) the number of mismatches allowed in these problems is usually small and (ii) we usually require a pair of approximately matched strings to share a long common substring (called a *common factor* in [61]). These algorithms have been carefully engineered to achieve satisfactory speeds on PCs, by taking into account the distribution of the frequencies of the words in the input unigene dataset. For example, running each of the algorithms for the barley unigene dataset from HARVEST takes only a couple of hours (on a 1.2 GHz AMD machine). This is a great improvement over other brute-force methods, like the ones based on BLAST. For example, one can identify unique oligos by repeatedly running BLAST for each unigene sequence against the entire dataset. This was the strategy previously employed by the HARVEST researchers. Simulations results show that the number of missed positives by the heuristic algorithm for popular oligos is very limited and can be controlled very effectively by adjusting the parameters.

In the context of DNA hybridization, most previous approaches define the specificity of an q-mer in terms of the number of mismatches to the target sequences, although some also take into account its physical and structural characteristics such as melting temperature, free-energy, GC-content, and secondary structure [52, 64]. In [61], Rahman took a more optimistic approach and used the length of the longest common substring (called the longest common factor or LCF) as a measure of specificity. Given the nature of our target applications, we will take a conservative approach in the definitions of unique and popular oligos.

3.1.2 Notations

We denote the input dataset as $\mathcal{X} = \{x_1, x_2, \ldots x_k\}$, where the generic string x_i is an unigene sequence over the alphabet $\Sigma = \{A, C, G, T\}$ and k is the cardinality of the set. Let n_i denote the length of the i-th sequence, $1 \leq i \leq k$. We set $n = \sum_{i=1}^{k} n_i$, which represents the total size of the input. A string (or oligo) from Σ is called an *q-mer* if its length is q.

Given a string A, we write $A[i]$, $1 \leq i \leq |A|$, to indicate the i-th symbol in A. We use $A[i, j]$ as a shorthand for the substring $A[i]A[i + 1] \ldots A[j]$ where $1 \leq i \leq j \leq n$, with the convention that $A[i, i] = A[i]$. Substrings in the form $A[1, j]$ correspond to the *prefixes* of A, and substrings in the form $A[i, n]$ to the *suffixes* of A. A string B *occurs* at position i of another string A if $B[1] = A[i], \ldots, B[l] = A[i + l - 1]$, where $l = |B|$.

Given two strings A and B of the same length, we denote by $H(A, B)$ the Hamming distance between A and B, that is, the number of mismatches between A and B.

3.2 Unique oligo

The unique oligo problem has been studied in the context of *probe design* [52, 61, 64]. The algorithms in [52, 64] consider physical and structural properties of oligos and are very time consuming. (The algorithm in [64] also uses BLAST.) The algorithm presented in [61] is, on the other hand, purely combinatorial. It uses suffix arrays instead of hash tables, and requires approximately 50 hours for a dataset of 40 Mb on a high-performance Compaq Alpha machine with 16 Gb of RAM. However, his definition of unique oligos is slightly

different from ours.

3.2.1 Definition

Recall that a unique oligo perfectly matches a unigene (called target) but does not match any other unigenes (called nontargets). Therefore, the definition of unique oligo depends on the criterion of whether the oligo candidate matches the nontarget sequences. We call the latter matching as "nontarget-match".

Definition 3.2.1 *Given a set of integer pairs $\mathcal{P} = \{(l_1, d_1), \ldots, (l_h, d_h)\}$, two strings A and B of equal length are said to* nontarget-match *each other, if there exists a pair $(l', d') \in \mathcal{P}$ and there exist a substring A' of A and a substring B' of B starting at the same position, such that:*

- $|A'| = |B'| = l'$*, and*
- $H(A', B') \leq d'$*, where $H(A', B')$ is the number of mismatches between A' and B'.*

If string A nontarget-matches a substring of a unigene X, then we say that A *nontarget-matches the unigene X*. For specific projects, the thresholds list $\mathcal{P}$ can be decided accordingly. For example, in [80], $\mathcal{P} = \{(16, 0), (20, 1), (24, 2), (30, 3), (36, 4)\}$. We call a pair of integers in $\mathcal{P}$ as one *condition* of nontarget-match.

Definition 3.2.2 *Given a set of unigenes and a definition of nontarget-match, a* unique oligo *is a string that appears exactly in one unigene but nontarget-matches none of the other unigenes.*

Note that in the above definition the length l of the oligos is not fixed. Theoretically, l can range from one to the maximum length of the unigenes. Practically, l ranges from 25 to 50 (e.g., $l = 36$ in [80]). Our algorithm assumes that all the output oligos are of equal length, which satisfies most practical needs.

3.2.2 Algorithm

We design an algorithm for finding unique oligos as follows. For each substring A in each unigene, we check whether A nontarget-matches any other unigene. To become a candidate oligo, the oligo should also "survive" the filtration step (to be described in Section 3.4). Since filtration takes much less time than checking nontarget-match and many candidates fail the filtration, we perform the filtration step first. In the rest of this section we focus on nontarget-match, and we assume the filtration has been done.

Our strategy is to prune the search space by q-gram filtration. The algorithm is based on the following observation. Assume that A and B are two l'-mers such that $H(A, B) \leq d$. Divide A and B into $t = \lfloor d/2 \rfloor + 1$ substrings. That is $A = A_1 A_2 \cdots A_t$ and $B = B_1 B_2 \cdots B_t$, where the length of each substring is $q = \lceil l'/t \rceil$, except possibly for the last one. In practice, one can always choose l' and d so that l' is a multiple of t and hence A and B can be decomposed into t substrings of length q, which we call *seeds* (also see Section 2.3 in Chapter 2). It is easy to see that since $H(A, B) \leq d$, at least one of the seeds of A has at most one mismatch with the corresponding seed of B. Each condition (l', d) is associated with a seed length. For multiple conditions, we must choose the shortest seed.

Based on this idea, we design an efficient two-phase algorithm. In the first phase, we index all seeds from the unigenes into a table, called *seed-table*, such that given a query seed s all the substrings in the unigenes that have at most one mismatch with s can be output efficiently. In the second phase, for every oligo candidate A, we locate every substring B in the unigenes that have one seed in common with A, using the seed-table. If A nontarget-matches B, then A is discarded. On average, the number of such substring B is much smaller than the size of the unigene database, and we thus improve running time.

Phase 1. (SEED-INDEX) We hash all q-mers (seeds) from the input unigenes into a dictionary (seed-table) with 4^q entries. (If 4^q cannot fit in the main memory, one could use a hash table of an appropriate size.) Each entry of the table points to a list of locations where the q-mer occurs in the unigene sequences. Using the table we can immediately locate identical seeds in the unigenes. We also collect seeds that have exactly one mismatch with each other as follows. For each table entry corresponding to a seed y, we record a list of other seeds that have exactly one mismatch with y, by looking up table entries that correspond to all the 1-mutants of y. This list is called a *mutant list* of y.

Phase 2. (UNIQUE-TEST) For each oligo candidate x from unigene A, we query all its seeds against the seed-table to locate a list of potential matches $y_1, \ldots, y_t$ in the unigenes other than A. Then we verify whether x nontarget-matches any of the y_i. The verification of each condition of nontarget-match (see Definition 3.2.1) can be done by counting the number of mismatches for every m-mer region between x and y_i.

In the practice of unigene data analysis, we also need to consider the reverse strand of

each unigene. It is easy to modify the above algorithm to take this into account without a significant increase in time complexity.

Time complexity. Assume that the total number of bases in the unigene database is n, and the length of seed is q. The time complexity of phase one is simply $\Theta(qn + 4^q)$, where the second term reflects the time needed to initialize the seed-table. If we insert seeds into the table in lexicographic order, and we consider the overlapping between two successive seeds, then the time can be easily reduced to $\Theta(n + 4^q)$.

The time complexity of phase two is $\Theta(nV)$, where V is the average time for verifying oligo candidates by checking nontarget-match. Clearly, V depends on the *filtering efficiency* of seeds (i.e., the number of matches found divided by the number of potential matches) and the time for verifying nontarget-match. We do not analyze the filtering efficiency here. But one need to make sure that the seed length q is not too small, otherwise there may be too many potential matches. On the other hand, q should not be too big, since the size of seed-table depends exponentially on q.

3.2.3 Group-unique oligo

A *unigene group* is a collection of unigene sequences that are similar with each other (e.g., they could originate from a gene family). It is useful to design oligos that are specific to a unigene group rather than an individual unigene. In [20], a method called Hierarchical Probe Design (HPD) is proposed for finding long oligos from conserved functional genes. HPD uses clustering methods based on pairwise comparison, e.g. UPGMA, neighbor-joining etc.

It has been tested on only small datasets of two types of genes of totally 911 sequences. It is easy to extend the idea for unique oligos to "group-unique oligos".

Definition 3.2.3 *For a given unigene group G from a set of unigenes X, a* group-unique oligo *is a DNA string that appear exactly in each unigenes in G, but does not nontarget-match any unigene of X not in G.*

The algorithm of designing group-unique oligos is similar to that for unique oligos, except that the candidates are from every unigenes of the group. We scan unigenes of a group and then cluster l-mer substrings into a table such that identical l-mers can be accessed by the same entry. Clearly this can be done in linear time. The rest is similar to unique oligo algorithm.

3.3 Popular oligo

The problem of finding infrequent and frequent patterns in sequences is a common task called pattern discovery. A quite large family of pattern discovery algorithms has been proposed in the literature and implemented in software tools. Without pretending to be exhaustive, we mention MEME [4], PRATT [44, 43], TEIRESIAS [62], CONSENSUS [40], GIBBS SAMPLER [50, 56], WINNOWER [58, 45], PROJECTION [72, 13], VERBUMCULUS [2], MITRA [26], among others. Although these tools have been demonstrated to perform very well on small and medium-size datasets, they cannot handle large datasets such as the barley unigene dataset that we are interested in. In particular, some of these tools were designed to

attack the "challenge" posed by Pevzner and Sze [58], which is in the order of a few Kb. Among the more general and efficient tools, we tried to run TEIRESIAS on the 28 Mb barley unigene dataset on an 1.2GHz Athlon CPU with 1GB of RAM, without being able to obtain any result (probably due to lack of memory).

3.3.1 Definition

Simply speaking, a popular oligo is a string that approximately matches as many unigenes as possible. Analogous to the Definition 3.2.1 of nontarget-match for unique oligos, we will define "target-match" between an oligo and a substring in unigenes to control the specificity of popular oligos. Again, we consider only mismatches. If the number of mismatches is too small, we may miss valuable oligos; if it is too big, we may get many spurious popular oligos. Therefore, the definition of target-match is crucial for the quality of output popular oligos.

In the following we define two versions of target-match, both of which require the presence of a perfectly matching segment called "core". For the rest of the section, we assume that strings A and B have equal length l.

Definition 3.3.1 *We say that there is a* core *of length l_c between A and B if A and B can be partitioned into substrings as $A = A_1A_2A_3$ and $B = B_1B_2B_3$ with $|A_i| = |B_i|$ such that (i) $|A_2| = l_c$, and (ii) $A_2 = B_2$.*

In the rest of the chapter, we assume that the length of core l_c is given. In [77, 78, 80], we have fixed $l_c = 20$.

The difference between the following two definitions of target-match lays in the number of mismatches allowed in the regions flanking the core. The first definition simply sets an upper bound for the total number of mismatches outside the core.

Definition 3.3.2 *We say that a string* A target-matches *a string* B *if (i) there is a core between them, and (ii)* $H(A, B) \leq d_{\max}$.

The second definition considers not only the number of mismatches but also their locations. The parameters are valid only for $l = 36$ ([80]). The threshold values should be changed for different values of l and l_c.

Definition 3.3.3 *We say that* A *and* B target-match *each other, if (i)there is a core between* A *and* B*, and (ii)either one of the following two conditions is satisfied*

- $H(A, B) < 3$*, or*
- $H(A, B) = 3$*, and*
  - *for any pair* A', B' *of* 25*-mers obtained by extending the core,* $H(A', B') < 2$*, and*
  - *for any pair* A'', B'' *of* 30*-mers obtained by extending the core,* $H(A'', B'') < 3$.

We call any string that target-match A as a *mutant* of A, and the set of all mutants of A as the *neighborhood* of A. If a popular oligo target-matches a substring in a unigene, we say that the oligo *covers* the unigene. Let $\mathcal{X}$ be the given set of unigenes. We call the number of distinct unigenes from $\mathcal{X}$ covered by A as the number of *colors* of A in $\mathcal{X}$, denoted $C_{\mathcal{X}}(A)$.

Definition 3.3.4 *Given a set of unigenes $\mathcal{X}$ and a positive integer $C_{\min}$, a* popular oligo *is a string A of length l such that $C_{\mathcal{X}}(A) \geq C_{\min}$.*

3.3.2 Algorithm

Since popular oligos are not required to appear exactly in the unigene sequences, the number of oligo candidates is much bigger than the size of unigenes. If we were to use Definition 3.3.2, the number of mutants of an oligo of length l is $\Theta(\binom{l-l_c}{d}3^d)$, where l_c is the core length and d the maximum number of mismatches. For example, when $d = 3$, $l = 33$, and $l_c = 20$, $\binom{l-l_c}{d}3^d = \binom{13}{3}3^3 = 7722$ for the barley dataset. Hence, the straightforward algorithm would have to count the number of colors for about $7722 \cdot 28 \times 10^6 = 217 \times 10^9$ l-mers. Therefore, the "brute-force" method that enumerates neighborhoods of l-mers in the unigenes is computationally impractical, due to its memory requirement as soon as the input size reaches the order of hundreds of thousands of bases (like the barley dataset).

We can reduce the search space using the same idea as in the algorithm for unique oligos, except that here the role of seeds is played by cores. Observe that if a popular oligo covers many unigenes, then many of these unigenes must contain length-l substrings that share common cores. Based on this observation, we propose a heuristic strategy that first clusters the l-mers in the unigene sequences into groups by their cores, and then enumerates candidate l-mers by comparing the members of each cluster in a hierarchical way.

An outline of the algorithm is illustrated in Figure 3.1. Here, we determine the popularity of the cores (*i.e.*, length-l_c substrings) from the unigenes in the first step. For each popular

core, we consider extension of the cores into l-mers by including flanking regions and cluster them using a well-known hierarchical clustering method, called *unweighted pair group method with arithmetic mean* (UPGMA) [71]. We recall that UPGMA builds the tree bottom-up in a greedy fashion by merging groups (or subtrees) of data points that have the smallest average distance. Based on the clustering tree, we compute the common oligos shared by the l-mers by performing set intersection. These common oligos shared by many l-mers become candidate popular oligos. Finally, we count the number of colors of these candidates, and output the oligos with at least T colors. A more detailed description is given below. A complete example of the algorithm on a toy dataset is also given at appendix.

Phase 1. We compute the number of colors for all l_c-mers in the unigenes to determine whether they could be candidate cores for popular l-mers, using a hash table. According to our definition, a popular oligo should have a popular core. We therefore set a threshold T_c on the minimum number of colors of each popular core, depending on $C_{\min}, l_c, l$ and the set of unigenes $\mathcal{X}$. All cores that have a number of colors below T_c are filtered out, and considered "unpopular". However, since an l-mer can target-match another l-mer with any of its $l - d + 1$ cores, it is possible that we might miss some popular oligos that critically depend on unpopular core. The parameter T_c represents a trade-off between precision and efficiency. We will show in Section 3.5 the effect of changing T_c on the output. We will see that in practice we might miss only a negligible number of popular oligos.

Phase 2. Here we collect the substrings flanking the popular cores. For each popular core, we construct $l - l_c + 1$ sets of substrings, one for each possible extension of the core

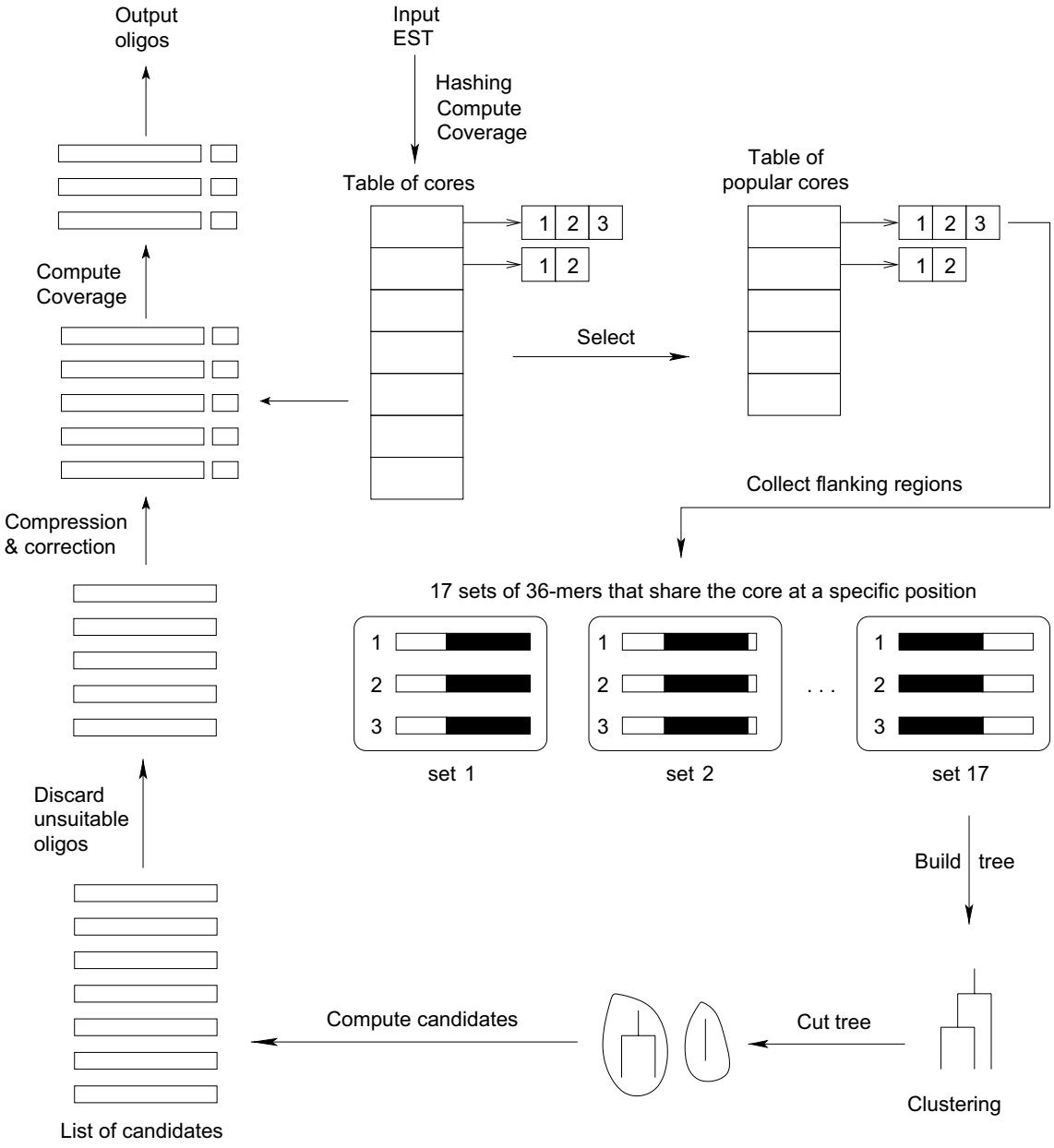

Figure 3.1: An overview of the algorithm for selecting popular oligos. For convenience of illustration, the length of the oligos is assumed to be $l = 36$, and the length of the cores is assumed to be $l_c = 20$.

into an l-mer.

Phase 3. For each set of extended l-mers, we would like to identify all l-mers (oligo candidates) that target-match many of these substrings. In order to achieve this efficiently,

we first cluster the substrings according to their mutual Hamming distance using the hierarchical clustering method UPGMA. In the process of building the clustering tree, whenever the Hamming distance between some leaves in the tree is zero we compress the distance matrix by combining the identical strings into one entry. This significantly reduces the running time not only because the tree becomes smaller, but also because the number of common mutants of two different l-mers is much less than that of two identical ones. As we can see later, a significant proportion of the running time is spent on the intersection of the sets of d-mutants. Compressing the distance matrices avoids intersecting identical sets of d-mutants, which is expensive and also useless. We then enumerate the set of mutants for each substring represented at the leaves and traverse the tree bottom-up.

At each internal node u, we compute the intersection of the two sets attached to the children. This intersection represents all the l-mers that target-match all the leaves (substrings) under the node u. As soon as the intersection of some internal node, say u, becomes empty, we cut the tree at u. Each subtree represents a cluster, and the set of l-mers attached to the root are the elements of the cluster. The size of the cluster is therefore equal to the number of leaves in the tree. Because small clusters are unlikely to contain popular oligos, we discard all trees whose size is smaller than $C_{\min}$ (the minimum number of colors of a popular oligo). At the end of this process, we obtain a collection of sets of candidate popular oligos.

Phase 4. Given the candidate popular oligos, we do the filtration step to discard unsuitable candidates and select a subset of oligos to maximize coverage rate. Because the oligo filtration and selection steps are necessary also for unique oligo, we will discuss these steps

in Section 3.4.

Time complexity. Phase 1 costs time $O(l_c n)$. In phase 2, if the number of popular cores selected in the first step is p and the average number of occurrence of the cores is r, this phase costs $O(nrl))$. For phase 3, the time for building a UPGMA tree, including the computation of the distance matrix, is $O((l-l_c)r^2)$, where r stands for the number of strings to be clustered. Since a (binary) UPGMA tree with r leaves has $2r-1$ nodes, the time for traversing (and pruning) the tree is $O(r\binom{l-l_c}{d}3^d)$, where $\binom{l-l_c}{d}3^d$ is the upper-bound of the number of mutants at each leaf, for both Definition 3.3.2 and Definition 3.3.3 of target-match. Here we assume that the set intersection can be done in linear time using hashing techniques. Finally for phase 4, if the total number of candidates is m, counting the colors for the candidates, excluding the time for radix-sort, costs time $O(rm(l-l_c))$.

3.4 Oligo filtration and selection

As it turns out, the set of unique, popular and group-unique oligos generated by the algorithms described above cannot be used directly because in practice it produces too many candidates. For example, when the threshold T_c on the color of the cores is 5, the number of candidates of popular oligos generated from the Barley unigene database is about 527 millions. In this section, we describe post-processing phase for reducing the number of candidate oligos.

In the first step of post-processing, called *oligo filtration*, we discard unsuitable oligos

based on GC content, melting temperatures, self-annealing of 36-mers, low-complexity, and the presence of repetitive regions. All these parameters can be adjusted by the user. The melting temperature T_m is calculated using the formula in [12] as implemented in PRIMER3 [65]. Self-annealing of oligos is determined by performing an end-free sequence alignment between the 22-mer prefix and the reverse complement of the 22-mer suffix of an oligo. An oligo is discarded if the alignment score is higher than a predetermined threshold. We use the program DUST [38] to determine low-complexity regions in oligos. Finally, we filter out those oligos that have significant matches against repeat database, e.g., Triticeae Repeat Sequence Database (TREP, `http://wheat.pw.usda.gov/ITMI/Repeats/`) in the case of barley, or any other repeat database provided by the user.

In the second step of post-processing, we select oligos according to their distribution among unigenes. Here, for a popular oligo, the "distribution" means the set of covered unigenes, while for a unique oligo it means the location within a unigene. Our objective is to select a small number of oligos that can represent as much information about unigenes as possible.

Unique oligo selection. The selected unique oligos should be as different as possible. Because a substring of length l can overlap $l - 1$ symbols with other substrings, oligos whose position is close to each other in a unigene are more likely to have similar composition. However, we prefer oligos that originate from diverse parts of unigene sequences. Therefore, the objective is to select a subset of oligos whose positions are as separated as possible.

We formulate the problem as a combinatorial optimization problem called *sparsest subsequence*: Given a sequence of integers $p_1, p_2, ..., p_n$ in increasing order, find a subsequence of size m, such that the minimum difference of two successive numbers in the subsequence is maximized.

An easy dynamic programming algorithm ([25]) can solve the problem in polynomial time. Let us call the minimum difference of two successive numbers as *gap*. Let $D[i, j]$ denote the maximum gap among the sequences of j numbers ending with P_i, for $1 \leq i \leq m$ and $1 \leq j \leq m$. We have the following recursive relation:

$$D[i, j] = \max_{k<i}(\min(D[k, j-1], P_i - P_k)).$$

By binary searching for k giving the maximum, we can solve the problem in time $O(nm \log n)$. The running time can be improved to $O(n + m \log n \log(n/m))$ ([18]). However, since algorithm in [18] is complicated and the sparsest subsequence problem is not the bottleneck of the whole system, we did not employ this faster algorithm.

Popular oligo selection. In general, each popular oligo covers a set of unigenes, and each unigene is covered by a set of oligos. For a set S of oligos, the *coverage rate* is the ratio between the number of unigenes covered by the oligos in S and the size of the set S. Our objective is to maximize the coverage rate, i.e., to select a set of popular oligos S from the large pool of candidates, such that the number of covered unigenes is maximized and the number of selected popular oligos is minimized. More specifically, while we are trying to

reduce the size of S, we make sure that the number of covered unigenes will not drop. It turns out that the general problem of oligo compression is a variant of the SET COVERING problem, which is known to be NP-complete (see [30]).

Since the general problem is NP-complete, it is unlikely that there exists a polynomial-time algorithm that finds the optimal solution. As a workaround, we use a greedy strategy that, in general, will find a suboptimal solution. The following paragraph is a rephrasing of the greedy algorithm for the SET COVERING ([42]) problem in the context of popular oligo selection.

In the first step, for each covered unigene we select a set of oligos with high colors, as follows. When a candidate oligo w is generated, we obtain the set of unigenes covered by w. For each covered unigenes, we decide whether oligo w should be discarded or kept as the top candidate. At the end of this step, we get a pool of unigenes each of which is covered by several oligos. In the second step, we select an oligo with the highest color, and remove all unigenes covered by this oligo from the unigene pool. Then, we update the colors of all other oligos. We repeat the second step until the unigene pool becomes empty. Note that it is important to update the colors iteratively because many candidates of high initial colors have big overlaps of covered unigenes with each other. The method is simple and space efficient since it can compress oligos on-line and avoid storing hundreds of millions of candidates in main memory.

3.5 Results

3.5.1 Implementation

We have implemented the above algorithms into the software OLIGOSPAWN, developed using the GNU C++ compiler under the Linux operating system. The executable for Linux/i386 can be downloaded from the OLIGOSPAWN web site. The source code is also available from the same web site under the GPL license. Any platform for which GNU C++ is available (Windows and MacOS among others) would be able to compile and run the stand-alone software. The web server is running at `http://oligospawn.ucr.edu/` and it was developed using PHP (`http://www.php.net/`), which is an open-source scripting language. The web server has been tested with Netscape, Mozilla, Safari, and Internet Explorer (see [80] for more details).

3.5.2 Simulations

Popular oligo. To evaluate the performance of our algorithm for designing popular oligos, we first ran a few simulations as follows. We generated a set of artificial unigenes by creating first a set of k random sequences and then inserting a controlled number of approximate occurrences of a given set of oligos, i.e., each of the inserted strings target-match one of the given oligos. We used the Definition 3.3.2 for target match, where the number of mismatches outside the core is at most d. The initial sets of oligos, denoted by $I_1, \ldots, I_s$, were also generated randomly. Each oligo I_i was assigned a predetermined number of colors C_i. We decided

	$d = 2$	$d = 3$
$T_c = 10$	0.0155	0.0500
$T_c = 15$	0.0003	0.0033
$T_c = 20$	0.0048	0.0005
$T_c = 25$	0.0008	0.0023
$T_c = 30$	0.0005	0.0028

Table 3.1: The average relative errors between the number of colors in the input and the number of colors in output for a simulated experiment ($n = 1.44 \times 10^6$, $t = 2000$, $l_c = 20$, $c_{\max} = 100$, $s = 100$).

that the distribution of the C_i should be Gaussian, i.e., we defined $C_i = c_{\max} e^{-i^2/2}/\sqrt{2\pi}$, where $c_{\max}$ is a fixed constant which determines the maximum number of colors. As said, the positions in-between the oligos were filled with random symbols over the DNA alphabet.

We then ran our program for popular oligos on the artificial unigenes dataset and output a set of candidate oligos $O_1, \ldots, O_t$ with their respective numbers of colors $C'_1, \ldots, C'_t$. The output oligos were sorted by colors, that is $C'_i \geq C'_j$, if $i < j$.

Since the output contained redundant candidates that came from the mutations of the original popular oligos, we removed those candidates that were mutants of another oligo with an higher number of colors. More precisely, if O_i was a mutant of O_j, and $1 \leq i < j \leq t$, then O_j was discarded. This "compression step" did not eliminate good candidates for the following reason. Since the input oligos $I_1, \ldots, I_s$ were generated randomly they were very unlikely to be similar. As a consequence, the corresponding output oligos were also unlikely to be eliminated. Moreover, the above extra step is unnecessary for real data, since a good oligos may not match exactly in unigenes. It is only for the convenience of validation

Finally, we compared the pair (I, C) with (O, C'). The more similar (O, C') is to (I, C),

the better is our algorithm. Recall that I and O were sorted by decreasing number of of colors. We compared the entries in (I, C) with the ones in (O, C'), position by position. For each $1 \leq i \leq u$, where $u = \min(s, t)$, we computed the average difference between C and C' as $E = (1/u) \sum_{i=1}^{u} \frac{|C_i - C'_i|}{C'_i}$. If we assume that I and O contain the same set of oligos, then the smaller is E, the more similar is (I, C) to (O, C'). To validate this assumption, we also searched the list of oligos I in O, to determine whether we missed completely some oligos.

Table 3.1 shows the value of E for four runs of the program on a dataset of $n = 1.44 \times 10^6$ bases composed by $t = 2000$ sequences each of size 720. We generated a set of $s = 100$ oligos with a maximum number of colors $c_{\max} = 100$. In the analysis, we fixed the length of the core to be $l_c = 20$, whereas the maximum number of mismatches d outside the core and the threshold T_c were varied. The results show that the average relative error is below 2%. We also compared the list of input oligos with the list of output oligos and we found that sometimes the program misses one or two oligos out of 100. However, the number of colors of these missed oligos is always near the threshold T_c. We never miss an oligo whose number of color is above $T_c + 10$.

Group-unique oligo. We also use BLAST to test the *oligo specificity* of group-unique oligos, where the specificity is the likelihood of matching nontarget unigenes. Since unique oligos can be regarded as a special case of group-unique oligos where each group consists of one unigene, the simulation can also be applied to unique oligos.

Running on a set $\mathcal{X}$ of unigenes, OLIGOSPAWN produces a list of oligos, where each oligo P corresponds to a set of unigenes U_p. Then, we query oligo P against $\mathcal{X}$ using BLAST.

E-value cut-off	1e-008	1e-007	1e-006	1e-005	0.0001	0.001
sensitivity(%)	95.9	95.4	94.9	94.2	93.2	90.4
selectivity(%)	99.2	99.6	99.8	99.9	100	100

Table 3.2: Specificity of group-unique oligo by OLIGOSPAWN validated by BLAST.

With a specified threshold of E-value, we obtain a list of unigenes in $\mathcal{X}$, denoted by V_p, which P covers by the matching criteria of BLAST. Let V_p denote the actual list of unigenes that oligo P covers. Then, we set the sensitivity as $\sum_p |U_p \cap V_p| / \sum_p |V_p|$ and selectivity as $\sum_p |U_p \cap V_p| / \sum_p |U_p|$.

In Table 3.2, each column corresponds to a threshold of BLAST e-value. Smaller e-value cut means that the BLAST matches between oligos and the target regions in unigenes are less likely to be random. Therefore, the bigger e-value cut means more oligo occurrences will be found by BLAST. When e-value cuts increase from left to right, the sensitivity of OLIGOSPAWN decreases and selectivity increases. When e-value cut is 1e-008, even exact matches between oligos and unigenes are considered not significant enough by BLAST. Therefore, the specificity of group-unique oligos designed by OLIGOSPAWN is high.

3.5.3 Running on real data

The main dataset is a collection barley unigenes from HARVEST. Before doing the searches, we first cleaned the dataset by removing PolyT and PolyA repeats.

The efficiency of our algorithm critically depends on the statistical distribution of the seeds in the dictionary. The statistics of the seeds in our experiment (before the extension

number of occurrences	number of seeds
0	242399
1-9	3063288
10-19	708745
20-29	120698
30-39	31637
40-49	11908
50-5049	15629

Table 3.3: Distribution of frequencies of seeds in barley unigenes. The left column is the range of the number of occurrences. The right column is the number of seeds with a certain number of occurrences.

phase) is shown in Table 3.3. Clearly, most seeds occur less than 20 times in the unigenes and this is the main reason why our algorithm was able to solve the dataset efficiently. The final distribution of unique oligos is shown in Figure 3.2.

Our second task was to search for popular oligos with length $l = 36$ and core length $l_c = 20$. We considered different choices for the maximum number of mismatches d outside the core and the threshold T_c on the minimum number of colors for the popular cores. The threshold $C_{\min}$ on the size the clusters was set equal to the value of threshold T_c.

The distribution of the number of colors of the cores is shown in Table 3.4. From the table we can see that the number of cores decreases almost exponentially as the number of colors increases. On the other hand, cores with low colors are unlikely to contribute to popular oligos. Therefore, it is important to filter them out to increase the efficiency.

The running time of this program varies with the parameters d and T_c, as shown in the Figure 3.3. The memory used in the program was mainly for storing the candidate popular oligos. In general, about 64 MB suffices since the program reuses the memory frequently.

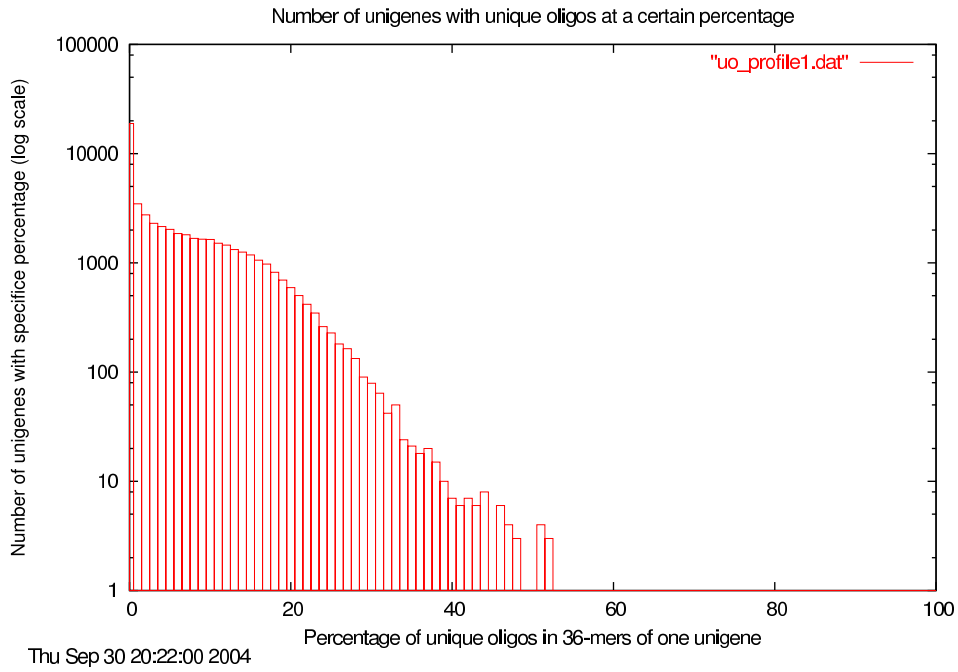

Figure 3.2: Distribution of unique oligos. The horizontal axis stands for the percentage of unique oligos over all 36-mers in a unigene, and the vertical axis stands for the number of unigenes whose unique oligos are at a certain percentage of all its 36-mers.

Figure 3.3 also shows the number of candidates generated by the algorithm (in millions), the number of unigenes covered, the final number of popular oligos, and the coverage ratio, for different choices of the threshold T_c.

3.5.4 Overgo hybridization

Popular 36-mer oligos were generated by an older version of the software OLIGOSPAWN with threshold $T_c = 4$, GC content in the range 45–56%. Since the older version of OLIGOSPAWN did not yet offer filtering against repeat databases this process was supplemented by some

colors	number of cores
1	22523412
2-10	2128677
11-20	5148
21-30	1131
31-40	492
41-50	346
51-60	242
61-70	77
71-80	34
81-90	29
91-100	43
101-176	19

Table 3.4: Distribution of the number of colors of the cores. The left column is the range of the number of colors. The right column is the number of cores with a certain number of color.

manual actions, as follows. Oligos matching repetitive DNA and rRNA were filtered out with BLAST searches (BLASTn) against TREP and the TIGR *Gramineae* repeat databases (*Hordeum, Oryza, Sorghum, Triticum, Zea*) (`http://www.tigr.org/tdb/e2k1/plant.repeats/` [57]). Following this search, 36-mers with 26 or more consecutive matches to repetitive sequences were discarded. Out of 698 initially proposed popular oligos, a total of 25 were discarded by this method. All these filtering step are now included in OLIGOSPAWN (in particular BLAST is not required to run OLIGOSPAWN).

The popular 36-mers were also "blasted" (by BLASTx) against the SwissProtein (`http://us.expasy.org/sprot/`) and NR protein databases for annotation purposes. The 36-mers with nine of twelve possible amino acids identical to the subject sequence were chosen for further testing. Out of the initial 698 popular 36-mers analyzed, 134 passed this criterion. Finally, popular oligos classified as transcription and signal transduction components,

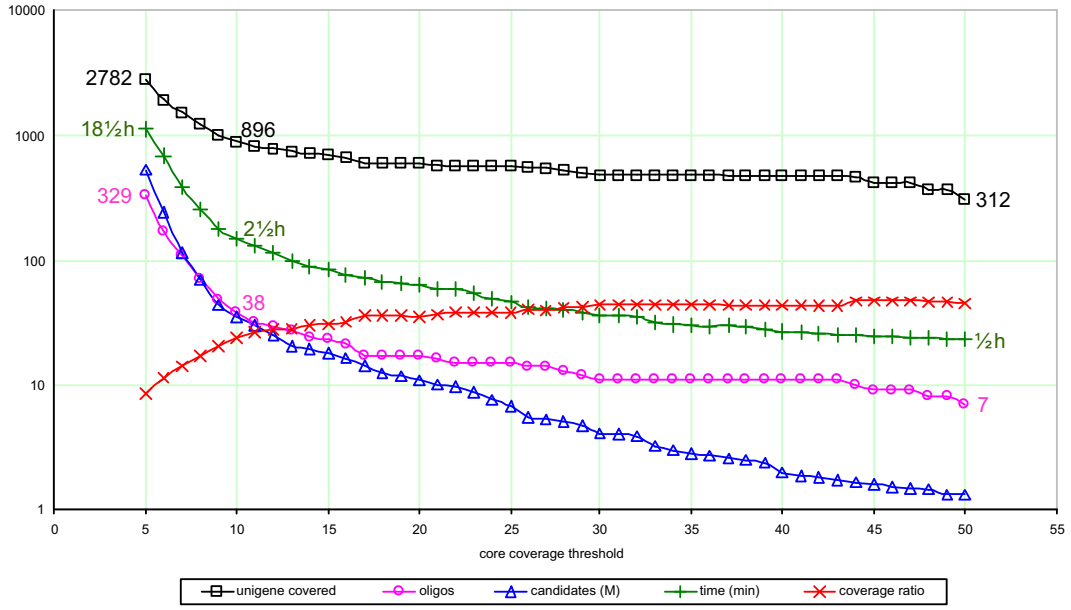

Figure 3.3: Results of running the algorithm on the Barley dataset. Shown are the number of candidates generated by the algorithm (in millions), the number of unigenes covered, the final number of popular oligos, the coverage ratio, and the time taken by the algorithm (for different choices of T_c).

a total of 18 out of these 134, were used for probing the Morex barley BAC library [76].

Overgo labeling and hybridization was done essentially as described by Ross *et al.* [63, 36]. Briefly, probes were radioactively labeled individually with ^{32}P-dATP and ^{32}P-dCTP. For background detection, a 36-mer representing the *Escherichia coli* genome was also labeled [36]. Hybridization using a mixture of all 19 probes was then performed on high-density filters of the 6.3$\times$ Morex barley BAC library [76], followed by washing and exposure to autoradiography film [63]. An average of 140 BAC clones per filter (17 filters) were scored as positive, yielding a total of about 2,400 positive BAC clones from only 18 popular overgos. Screening with 18 unique overgos would be expected to identify only about 113 total clones (17×6.3). Therefore, the 18 popular oligos described above netted about 22 times as many positive

clones as would unique oligos. Results with other sets of popular oligos not described in this manuscript have given comparable results. Therefore, we conclude that the popular oligo algorithm provides a substantial gain of efficiency in probing BAC genomics libraries for gene-containing clones.

The number of positive BAC clones identified with various pools sizes of unique oligos has consistently been in the range of 6 to 8 BACs per unique oligo. For example, pools of 192 unique oligos repeatedly provide about 1200 to 1600 positive BAC addresses. Furthermore, checking the sequences of unique oligos with BLAST has consistently provided assurance that our unique oligo algorithm indeed is as selective as it is intended to be.

Chapter 4

Minimum Common String Partition

In the previous two chapters, we presented approaches for finding repetitive patterns in biological sequences. In this chapter, we aim to infer evolutionary and functional relations among the discovered repeats. In the wide range of problems in this domain, we address the problem of genome rearrangement with multigene families, where duplicated genes are treated as repetitive patterns. In particular, we address the minimum common string partition problem (MCSP), which has tight connection with the problem of sorting by reversals with duplicates, a key problem in genome rearrangement. The restricted version of MCSP where each letter occurs at most k times in each input string is denoted by k-MCSP.

We show in Section 4.2 that 2-MCSP (and therefore MCSP) is NP-hard and, moreover, even APX-hard. Section 4.3 presents a 1.5-approximation and a 1.1037-approximation for 2-MCSP. This chapter is mainly from [31, 32], except that Section 4.3.1 is from [16].

4.1 Preliminaries

String comparison is a fundamental problem in computer science, with applications in areas such as computational biology, text processing and compression. Typically, a set of string operations is given (e.g., delete, insert and change a character, move a substring or reverse a substring) and the task is to find the minimum number of operations needed to convert one string to the other. Edit distance or permutation sorting by reversals are two well known examples. In this chapter we address, motivated mainly by genome rearrangement applications, the minimum common string partition problem (MCSP). Though MCSP takes a static approach to string comparison, it has tight connection to the problem of sorting by reversals with duplicates, a key problem in genome rearrangement.

A *partition* of a string A is a sequence $\mathcal{P} = (P_1, P_2, \ldots, P_m)$ of strings whose concatenation is equal to A, that is $P_1 P_2 \ldots P_m = A$. The strings P_i are called the *blocks* of $\mathcal{P}$. Given a partition $\mathcal{P}$ of a string A and a partition $\mathcal{Q}$ of a string B, we say that the pair $\pi = \langle \mathcal{P}, \mathcal{Q} \rangle$ is a *common partition* of A and B if $\mathcal{Q}$ is a permutation of $\mathcal{P}$. The *minimum common string partition* problem is to find a common partition of A, B with the minimum number of blocks. The restricted version of MCSP where each letter occurs at most k times in each input string, is denoted by k-MCSP. We denote by *#blocks*(π) the number of blocks in a common partition π. We say that two strings A and B are *related* if every letter appears the same number of times in A and B. Clearly, a necessary and sufficient condition for two strings to have a common partition is that they are related.

The *signed* minimum common string partition problem (SMCSP) is a variant of MCSP in which each letter of the two input strings is given a "$+$" or "$-$" sign (in genome rearrangement problems, the letters represent different genes on a chromosome and the signs represent orientation of the genes). For a string P with signs, let $-P$ denote the reverse of P, with each sign flipped. A common partition of two signed strings A and B is the pair $\pi = \langle \mathcal{P}, \mathcal{Q} \rangle$ of a partition $\mathcal{P} = (P_1, P_2, \ldots, P_m)$ of A and a partition $\mathcal{Q} = (Q_1, Q_2, \ldots, Q_m)$ of B together with a permutation σ on $[m]$ such that for each $i \in [m]$, either $P_i = Q_{\sigma(i)}$, or $P_i = -Q_{\sigma(i)}$.

New results. In this paper, we show that 2-MCSP (and therefore MCSP) is NP-hard and, moreover, even APX-hard. We also describe a 1.1037-approximation for 2-MCSP and a linear time 4-approximation algorithm for 3-MCSP. All of our results apply also to signed MCSP. We are not aware of any better approximations.

4.1.1 Related work

The problem of 1-MCSP coincides with the breakpoint distance problem of two permutations [75] which is to count the number of ordered pairs of symbols that are adjacent in the first string but not in the other; this problem is obviously solvable in polynomial time. Similarly as the breakpoint distance problem does, most of the rearrangement literature works with the assumption that a genome contains only one copy of each gene. Under this assumption, a lot of attention was given to the problem of sorting by reversals. *Reversal* is an operation that reverses a specified substring of a given string; in the case of signed strings,

it also flips the sign of each letter in the reversed substring. In the problem of *sorting by reversals*, the task is to determine the minimum number of reversals that transform a given string A into a given string B. The problem is solvable in polynomial time for signed strings containing only one copy of each symbol [39] but is NP-hard for unsigned strings [15]. The assumption about uniqueness of each gene is unwarranted for genomes with multi-gene families such as the human genome [67]. Chen et al. [16] studied a generalization of the problem, the problem of *signed reversal distance with duplicates* (SRDD); according to them, SRDD is NP-hard even if there are at most two copies of each gene. They also introduced the signed minimum common partition problem as a tool for dealing with SRDD. Chen et al. observe that for any two related signed strings A and B, the size of a minimum common partition and the minimum number of reversal operations needed to transform A to B, are within a multiplicative factor 2 of each other. (In the case of unsigned strings, no similar relation holds: the reversal distance of $A = 1234\ldots n$ and $B = n\ldots4321$ is 1 while the size of minimum common partition is $n - 1$.) They give a 1.5-approximation algorithm for 2-MCSP (see Subsection 4.3.1 for details), and use the algorithm to approximate SRDD. Christie and Irving [17] consider the problem of (unsigned) reversal distance with duplicates (RDD) and prove that it is NP-hard even for strings over binary alphabet.

Chrobak et al. [19] analyze a natural heuristic for MCSP, the greedy algorithm: iteratively, at each step extract a longest common substring from the input strings. They show that for 2-MCSP, the approximation ratio is exactly 3, for 4-MCSP the approximation ratio is $\Omega(\log n)$; for the general MCSP, the approximation ratio is between $\Omega(n^{0.43})$ and $O(n^{0.67})$.

The same bounds apply for SMCSP. In [31] Kolman gives a linear time 4-approximation algorithm for 3-MCSP. In [47] Kolman describes a simple modification of the greedy algorithm; the approximation ratio of the modification is $O(k^2)$ for k-MCSP and it runs in time $O(k \cdot n)$. The same bounds hold also for k-SMCSP and k-SRDD. Recently, [46] presents a linear time $\Theta(k)$-approximation algorithm for k-MCSP.

Closely related is the problem of edit distance with moves in which the allowed string operations are the following: insert a character, delete a character, move a substring. Cormode and Muthukrishnan [22] describe an $O(\log n \log^* n)$-approximation algorithm for this problem. Shapira and Storer [69] observed that restriction to move-a-substring operations only (instead of allowing all three operations listed above) does not affect the edit-distance of two strings by more than a constant multiplicative factor. Since the size of a minimum common partition of two strings and their distance with respect to move-a-substring operations differ only by a constant multiplicative factor, the algorithm of Cormode and Muthukrishnan yields an $O(\log n \log^* n)$-approximation for MCSP.

4.1.2 Combinatorial properties of MCSP

Throughout the chapter, we assume that the two strings A, B given as input to MCSP are related. This is a necessary and sufficient condition for the existence of a common partition.

Given a string $A = a_1 \dots a_n$, for the sake of simplicity we will use the symbol a_i to denote two different things. First, a_i may denote the specific occurrence of the letter a_i in the string A, namely the occurrence on position i. Alternatively, a_i may denote just the letter

itself, without any relation to the string A. Which alternative we mean will be clear from context.

Common partitions as mappings. Given two strings $A = a_1 \dots a_n$ and $B = b_1 \dots b_n$ of length n, a common partition π of A and B can be naturally interpreted as a bijective mapping from A to B (that is, if $P_1, \dots, P_m$ is the partition of A and $Q_1, \dots, Q_m$ is the partition of B in π, then for each $j \in [m]$, the letters from P_j are mapped from left to right to the corresponding $Q_{j'}$), and this in turn as a permutation on $[n]$. With this understanding in mind, we say that a pair of consecutive positions $i, i+1 \in [n]$ is a *break* of π in A if $\pi(i+1) \neq \pi(i)+1$. In other words, a break is a pair of letters that are consecutive in A but are mapped by π to letters that are not consecutive in B. The number of breaks in π will be denoted by *#breaks*(π).

Clearly, not every permutation on $[n]$ corresponds to a common partition of A and B. We say that a permutation ρ on $[n]$ *preserves letters* of A and B, if $a_i = b_{\rho(i)}$, for all $i \in [n]$. Then, every letter-preserving mapping ρ can be interpreted as a common partition ρ, and *#blocks*$(\rho) =$ *#breaks*$(\rho) + 1$. On the other hand, for a common partition $\pi = \langle \mathcal{P}, \mathcal{Q} \rangle$ interpreted as a permutation, *#blocks*$(\pi) \geq$ *#breaks*$(\pi) + 1$ (the inequality is due to possible unnecessary breaks in π). Thus, the MCSP problem is to find a permutation π on $[n]$ that preserves letters of A and B and has the minimum number of breaks. An alternative formulation is that the goal is to find a letter-preserving permutation that maps the maximum number of pairs of consecutive letters in A to pairs of consecutive letters in B.

Common partitions and independent sets. Let Σ denote the set of all letters that occur in A. A *duo* is an ordered pair of letters $xy \in \Sigma^2$ that occur consecutively in A or B (that is, there exists an i such that $x = a_i$ and $y = a_{i+1}$, or $x = b_i$ and $y = b_{i+1}$). A *specific* duo is an occurrence of a duo in A or B. The difference is that a duo is just a pair of letters whereas a specific duo is a pair of letters together with its position. A *match* is a pair $(a_i a_{i+1}, b_j b_{j+1})$ of specific duos, one from A and the other one from B, such that $a_i = b_j$ and $a_{i+1} = b_{j+1}$. Two matches $(a_i a_{i+1}, b_j b_{j+1})$ and $(a_k a_{k+1}, b_l b_{l+1})$, $i \leq k$, are *in conflict* if either $i = k$ and $j \neq l$, or $i + 1 = k$ and $j + 1 \neq l$, or $i + 1 < k$ and $\{j, j + 1\} \cap \{l, l + 1\} \neq \emptyset$. Informally, two matches are in conflict if they cannot be realized at the same time.

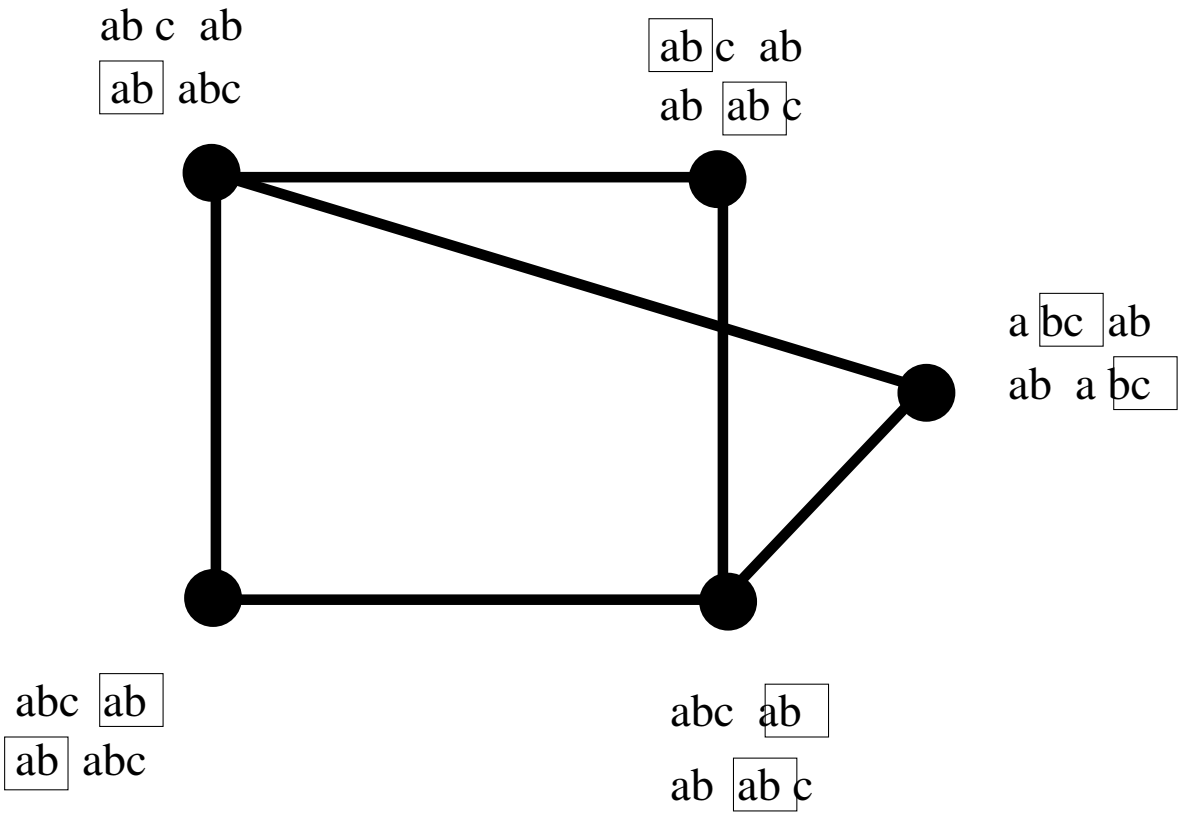

Figure 4.1: Conflict graph for MCSP instance $A = abcab$ and $B = ababc$.

We construct a *conflict graph* $G = (V, E)$ of A and B as follows. The set of nodes V consists of all matches of A and B and the set of edges E consists of all pairs of matches that are in conflict. Figure 4.1 shows an example of a conflict graph. The number of vertices in G can be much higher than the length of the strings A and B (and is trivially bounded by n^2).

Lemma 4.1.1 *For $A = a_1 \dots a_n$ and $B = b_1 \dots b_n$, let $MIS(G)$ denote the size of the maximum independent set of the conflict graph G of A and B and m denote the number of blocks in a minimum common partition of A and B. Then, $n - MIS(G) = m$.*

Proof. Given an optimal solution for MCSP, let S be the set of all matches that are used in this solution. Clearly, S is an independent set in G and $|S| = n - 1 - (m - 1)$.

Conversely, given a maximum independent set S, we cut the string A between a_i and a_{i+1} for every specific duo $a_i a_{i+1}$ that does not appear in any match in S, and similarly for B. In this way, $n - 1 - |S|$ duos are cut in A and also in B, resulting in $n - |S|$ blocks of A and $n - |S|$ blocks of B. Clearly, the blocks from A can be matched with the blocks from B, and therefore $m \leq n - |S|$. □

Maximum independent set is an NP-hard problem, yet, two approximation algorithms for MCSP described in this paper make use of this reduction.

MCSP **for multisets of strings.** For the proofs in later sections we need a slight generalization of the MCSP. Instead of two strings A, B, the input consists of two multisets $\mathcal{A}, \mathcal{B}$ of strings. Similarly as before, a partition of the multiset $\mathcal{A} = \{A_1, \dots, A_l\}$ is a sequence of

strings

$$A_{1,1}, \ldots, A_{1,k_1}, A_{2,1}, \ldots, A_{2,k_2}, \ldots, A_{l,1}, \ldots, A_{l,k_l},$$

such that $A_i = A_{i,1} \ldots, A_{i,k_i}$ for $i \in [l]$. For two multisets of strings, the common partition, the minimum common partition and the related-relation are defined similarly as for pairs of strings.

Let $\mathcal{A} = \{A_1, \ldots, A_l\}$ and $\mathcal{B} = \{B_1, \ldots, B_h\}$ with $h \leq l$, be two related multisets of strings, and let $x_1, y_1, \ldots, x_{l-1}, y_{l-1}$ be $2l-2$ different letters that do not appear in $\mathcal{A}$ and $\mathcal{B}$. Considering two strings

$$\begin{aligned} A &= A_1 x_1 y_1 A_2 x_2 y_2 A_3 \ldots x_{l-1} y_{l-1} A_l \,, \\ B &= B_1 y_1 x_1 B_2 y_2 x_2 B_3 \ldots y_{h-1} x_{h-1} B_h y_h x_h \ldots y_{l-1} x_{l-1} \,, \end{aligned} \tag{4.1}$$

it is easy to see that an optimal solution for the classical MCSP instance A, B yields an optimal solution for the instance $\mathcal{A}, \mathcal{B}$ of the multiset version, and vice versa. In particular, if m' denotes the size of a MCSP of the two multisets of strings $\mathcal{A}$ and $\mathcal{B}$, and m denotes the size of a MCSP of the two strings A and B defined as above, then

$$m = m' + 2(l-1) \,. \tag{4.2}$$

Thus, if one of the variants of the problems is NP-hard, so is the other.

4.2 Hardness of approximation

The main result of this section is the following theorem.

Theorem 4.2.1 2-MCSP *and* 2-SMCSP *are APX-hard problems.*

We start by proving a weaker result.

Theorem 4.2.2 2-MCSP *and* 2-SMCSP *are NP-hard problems.*

Proof. Since an instance of MCSP can be interpreted as an instance of SMCSP with all signs positive, and since a solution of SMCSP with all signs positive can be interpreted as a solution of the original MCSP and vice versa, it is sufficient to prove the theorems for MCSP only.

The proof is by reduction from the maximum independent set problem on cubic graphs (3-MIS) [30]. Given a cubic graph $G = (V, E)$ as an input for 3-MIS, for each vertex $v \in V$ we create a small instance I_v of 2-MCSP. Then we process the edges of G one after another, and, for each edge $(u, v) \in E$, we locally modify the two small instances I_u, I_v. The final instance of 2-MCSP, denoted by I_G, is the union of all the small (modified) instances I_v. We will show that a minimum common partition of I_G yields easily a maximum independent set in G.

The small instance $I_u = (X_u, Y_u)$ for a vertex $u \in V$ is defined as follows (cf. Figure 4.2):

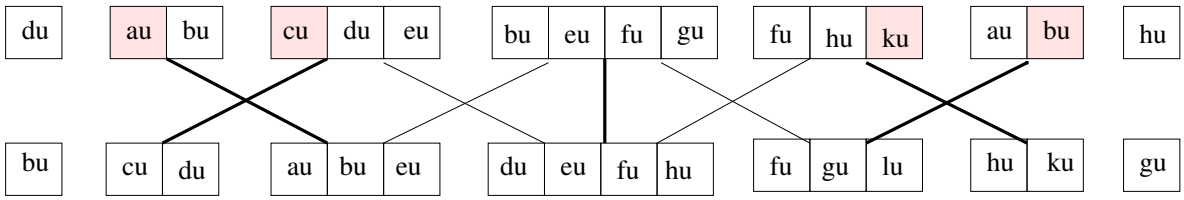

Figure 4.2: An instance I_u in the proof of NP-hardness of 2-MCSP. The lines represent all matches, with the bold lines corresponding to the matches in the minimum common partition O_u.

$$X_u = \{d_u,\ a_u b_u,\ c_u d_u e_u,\ b_u e_u f_u g_u,\ f_u h_u k_u,\ g_u l_u,\ h_u\} \tag{4.3}$$

$$Y_u = \{b_u,\ c_u d_u,\ a_u b_u e_u,\ d_u e_u f_u h_u,\ f_u g_u l_u,\ h_u k_u,\ g_u\}$$

where all the letters in the set $\cup_{u \in V}\{a_u, b_u, \ldots, l_v\}$ are distinct. It is easy to check that I_u has a unique minimum common partition, denoted by O_u, namely:

$$\begin{aligned} O_u \ = \ & \langle (d_u,\ a_u b_u,\ c_u d_u,\ e_u,\ b_u,\ e_u f_u,\ g_u,\ f_u,\ h_u k_u,\ g_u l_u,\ h_u) \\ & (b_u,\ c_u d_u,\ a_u b_u,\ e_u,\ d_u,\ e_u f_u,\ h_u,\ f_u,\ g_u l_u,\ h_u k_u,\ g_u)\rangle \end{aligned}$$

We observe that for $X_G = \bigcup_{u \in V} X_u$ and $Y_G = \bigcup_{u \in V} Y_u$, $I_G = (X_G, Y_G)$ is an instance of 2-MCSP, and the superposition of all O_u's is a minimum common partition of I_G. For the sake of simplicity, we will sometimes abuse the notation by writing $I_G = \bigcup_{u \in V} I_u$.

The main idea of the construction is to modify the instances I_u, such that for every edge $(u, v) \in E$, a minimum common partition of $I_G = \bigcup_{u \in V} I_u$ coincides with at most one of

the minimum common partitions of I_u and I_v. This property will make it possible to obtain a close correspondence between maximum independent sets in G and minimum common partitions of I_G: if O_v denotes a minimum common partition of (the modified) I_v and O'_v denotes the common partition of (the modified) I_v derived from a given minimum common partition of I_G, then $U = \{u \in V \mid O'_u = O_u\}$ will be a maximum independent set of G. To avoid the need to use different indices, we use I_G to denote $\bigcup_{u \in V} I_u$ after any number of the local modifications; it will always be clear from context to which one are we referring.

For description of the modifications, a few terms will be needed. The letters a_u and c_u in X_u are called *left sockets of* I_u and the letters k_u and l_u in X_u are *right sockets*. We observe that all the four letters a_u, c_u, k_u, l_u appears only once in X_G (and once in Y_G). Given two small instances I_u and I_v and a socket s_u of I_u and a socket s_v of I_v, we say that the two sockets s_u and s_v are *compatible*, if one of them is a left socket and the other one is a right socket. Initially, all sockets are *free*.

For technical reasons, we orient the edges of G in such a way that each vertex has at most two incoming edges and at most two outgoing edges. This can be done as follows: find a maximal set (with respect to inclusion) of edge-disjoint cycles in G, and in each cycle, orient the edges to form a directed cycle. The remaining edges form a forest. For each tree in the forest, choose one of its nodes of degree one to be the root, and orient all edges in the tree away from the root. This orientation will clearly satisfy the desired properties.

We are ready to describe the local modifications. Consider an edge $\overrightarrow{(u, v)} \in E$ and a free right socket s_u of I_u and a free left socket s_v of I_v. That is, $Rs_u \in X_u$ and $s_v S \in X_v$, for

some strings R and S. We modify the instances $I_u = (X_u, Y_u)$ and $I_v = (X_v, Y_v)$ as follows

$$\begin{aligned} &X_u \leftarrow X_u \cup \{Rs_uS\} - \{Rs_u\} \ , \quad && X_v \leftarrow X_v \cup \{s_u\} - \{s_vS\} \ , \\ &Y_u \leftarrow Y_u \ , && Y_v \leftarrow Y_v \text{ with } s_v \text{ renamed by } s_u \end{aligned} \tag{4.4}$$

(the symbols $\cup$ and $-$ denote multiset operations).

After this operation, we say that the right socket s_u of I_u and the left socket s_v of I_v are *used* (not free). Note that in Y_v, the letter s_v is renamed to s_u. All other sockets of I_u and all other sockets of I_v that were free before the operation remain free. We also note that I_u and I_v are not 2-MCSP instances. However, for every letter, the number of its occurrences is the same in X_G and in Y_G, namely at most two. Thus, I_G is still a 2-MCSP instance.

The complete reduction from a cubic graph $G = (V, E)$ to a 2-MCSP instance is done by performing the local modifications (4.4) for all edges in G.

Reduction of 3-MIS to 2-MCSP

1. $\forall u \in V$, define I_u by the description (4.3),
2. $\forall \overrightarrow{(u, v)} \in E$, find a free right socket s_u of I_u and a free left socket s_v of I_v,

   modify I_u and I_v by the description (4.4),
3. set $I_G = \bigcup_{u \in V} I_u$.

Since the in-degree and the out-degree of every node is bounded by two, and since every instance I_u has initially two right and two left sockets, there will always be the required free sockets.

It remains to prove that a minimum common partition for the final I_G (that is, when modifications for all edges are done) can be used to find a maximum independent set in G.

Lemma 4.2.1 *Let G be a cubic graph on N vertices. Then, there exists an independent set I of size h in G if and only if there exists a common partition of I_G of size $12N - h$.*

Proof. Let G_C be the conflict graph of I_G; G_C has $9N$ vertices. Let $O'_u = \{(d_uc_u, d_uc_u), (b_ue_u, b_ue_u), (f_ug_u, f_ug_u), (f_uh_u, f_uh_u)\}$, that is, O'_u is a set consisting of four out of the nine possible matches in the small instance I_u (in Figure 4.2, these four matches are represented by the thin lines). The crucial observation is that $\bigcup_{u \in V} O'_u$ is an independent set of size $4N$ in the conflict graph G_C.

Given an independent set I of G, construct a common partition of I_G as follows. For $u \in I$, use the five matches from O_u, and for $u \notin I$, use the four matches from O'_u. The resulting solution will use $5h+4(N-h)$ matches which corresponds to $9N-(5h+4(N-h)) = 5N-h$ new breaks and $7N + 5N - h = 12N - h$ blocks.

Conversely, given a common partition of I_G of size m, let I consist of all vertices u such that I_u contributes 5 matches (i.e., 11 blocks) to the common partition. Then, $h \geq 12N - m$, and the proof is completed. □

Since the reduction can clearly be done in polynomial time (even in linear), with respect to $|V|$ and $|E|$, the proof of NP-hardness of 2-MCSP is completed. □

Proof. (Theorem 4.2.1) We use the same construction and only complement calculations of the inapproximability ratio. Given a cubic graph G on N vertices, let m' denote the size

of a minimum common partition of the instance $I_G = (X_G, Y_G)$ and let m denote the size of a minimum common partition of the instance (A, B), derived from the multiset instance (X_G, Y_G) by relation (4.1). We note that each of X_G and Y_G consists of $7N$ strings. By Lemma 4.2.1 the size of a maximum independent set in G is $12N - m'$ which equals to $26N - 2 - m$ by relation (4.2) and the above observation about size of X_G and Y_G; thus, an α-approximation algorithm for MCSP on the instance (A, B) can be used to derive an independent set in G of size at least $26N - 2 - \alpha \cdot m$.

Berman and Karpinski [10] proved that it is NP-hard to approximate 3-MIS within $\frac{140}{139} - \epsilon$, for every $\epsilon > 0$. Thus, unless P=NP, for every $\epsilon > 0$, the approximation ratio α of any algorithm for MCSP must satisfy

$$\frac{26N - 2 - m}{26N - 2 - \alpha \cdot m} \geq \frac{140}{139} - \epsilon \, .$$

Solving for α yields, for every $\epsilon' > 0$,

$$\alpha \geq \frac{26N - 2 + 139m}{140m} - \epsilon' = 1 + \frac{26N - 2 - m}{140m} - \epsilon' \, .$$

Using the fact that a maximum independent set in any cubic graph on N vertices has always size at least $N/4$, we have $m \leq 26N - 2 - N/4$ and we conclude that it is NP-hard to approximate MCSP within $1 + \frac{1}{103 \cdot 140} - \epsilon$, for every $\epsilon > 0$. □

Remark: To prove that only SMCSP is APX-hard, it is possible to start with smaller instances I_u and thus get the constant larger.

4.3 Algorithms

4.3.1 Simple 1.5-approximation for 2-MCSP

We observed in Section 4.1 that MCSP can be restated as finding a maximum independent set in the conflict graph G which is the same as finding a minimum vertex cover (MVC) for G. Unfortunately, both these problems are NP-complete. Even worse, an approximation for the vertex cover does not transfer in general to an approximation for MCSP (and there are no good approximations for MIS). The problem is that there is no direct relation between the number of vertices in the MVC and the number of breaks (or blocks) in MCSP (while the size of MIS equals the number of unbroken duos in MCSP). There may be many vertices in MVC of G and still no breaks in MCSP. Fortunately, the situation is a bit easier for 2-MCSP.

In this and the following subsection we will assume that no duo appears at the same time twice in A and twice in B. The point is that in 2-MCSP, the minimum common partition never has to break such a duo. Thus, if there exists in A and B such a duo, it is possible to replace it by a new letter, solve the modified instance and then replace the new letter back by the original duo.

Given the assumption, we observe that the conflict graph $G = (V, E)$, for any 2-MCSP instance A, B on strings of length n, will have at most n vertices. The reason is that if a

pair $a_i a_{i+1}$ from A matches with two pairs from B, say with $b_j b_{j+1}$ and with $b_l b_{l+1}$, that the other occurrence of letter a_i in A, say at position a_k, is not followed by the other occurrence of a_{i+1}, and thus, the pair $a_k a_{k+1}$ cannot be matched with anything from B. Therefore, on average, every letter can appear in at most one match, and G has at most n vertices.

Consider now an α-approximation C for minimum vertex cover on the conflict graph $G = (V, E)$ and let C^* denote a minimum vertex cover. Then, by Lemma 4.1.1, a minimum common partition has $n - |V| + |C^*|$ blocks while a common partition corresponding to the vertex cover C has $n - |V| + |C|$ blocks. Exploiting the earlier observation that $|V| \leq n$, we get:

$$\frac{n - |V| + |C|}{n - |V| + |C^*|} \leq 1 + \frac{|C| - |C^*|}{|C^*|} \leq \alpha$$

Theorem 4.3.1 *An α-approximation algorithm for* MVC *yields an α-approximation for the* 2-MCSP.

Thus, a trivial 2-approximation for minimum vertex cover can be turned to a 2-approximation for 2-MCSP. Observing further that, the conflict graph is 6-claw free, for 2-MCSP, we can use 1.5-approximation algorithm for vertex cover by Halldórsson [35].

Corollary 4.3.2 *There exists a polynomial 1.5-approximation algorithm for* 2-MCSP *problem.*

4.3.2 Reducing 2-MCSP to MIN 2-SAT

In this section we will see how to solve 2-MCSP using algorithms for MIN 2-SAT. We start by recalling the definition of MIN 2-SAT problem. In MIN 2-SAT we are given a boolean formula in conjunctive normal form such that each clause consists of at most two literals, and we seek seek an assignment of boolean values to the variables that minimizes the number of satisfied clauses. Avidor and Zwick [3] proved that unless P=NP, the problem cannot be approximated within $15/14 - \epsilon$, for any $\epsilon > 0$, and they also gave a 1.1037-approximation algorithm which is the best approximation algorithm for the problem we are aware of. The main result of this section is stated in the following theorem.

Theorem 4.3.3 *An α-approximation algorithm for* MIN 2-SAT *yields α-approximations for both* 2-MCSP *and* 2-SMCSP.

Corollary 4.3.4 *There exist polynomial* 1.1037*-approximation algorithms for* 2-MCSP *and* 2-SMCSP *problems.*

Proof. (Theorem 4.3.3) There are only minor differences between the reductions for signed and unsigned versions of the problem. We describe in detail the reduction for 2-MCSP and then briefly point out the differences for 2-SMCSP.

Let A and B be two related strings. We start the proof with two assumptions that will simplify the presentation:

(1) no duo appears at the same time twice in A and twice in B, and that

(2) every letter appears exactly twice in both strings.

Concerning the first assumption, the point is that in 2-MCSP, the minimum common partition never has to break such a duo. Thus, if there exists in A and B such a duo, it is possible to replace it by a new letter, solve the modified instance and then replace the new letter back by the original duo. Concerning the other, a letter that appears only once can be replaced by two copies of itself. A minimum common partition never has to use a break between these two copies, so they can be easily replaced back to a single letter, when the solution for the modified instance is found.

The main idea of the reduction is to represent a common partition of A and B as a truth assignment of a (properly chosen) set of binary variables. With each letter $a \in \Sigma$ we associate a binary variable X_a. For each letter $a \in \Sigma$, there are exactly two ways to map the two occurrences of a in A onto the two occurrences of a in B: either the first a from A is mapped on the first a in B and the second a from A on the second a in B, or the other way round. In the first case, we say that a is mapped *straight*, and in the other case that a is mapped *across*. Given a common partition π of A and B, if a letter $a \in \Sigma$ is mapped straight we set $X_a = 1$, and if a is mapped across we set $X_a = 0$. In this way, every common partition can be turned into truth assignment of the variables X_a, $a \in \Sigma$, and vice versa. Thus, there is one-to-one correspondence between truth-assignments for the variables X_a, $a \in \Sigma$, and common partitions (viewed as mappings) of A and B.

With this correspondence between truth assignments and common partitions, our next goal is to transform the two input strings A and B into a boolean formula φ such that

- φ is a conjunction of disjunctions (OR) and exclusive disjunctions (XOR),
- each clause contains at most two literals, and
- the minimum number of satisfied clauses in φ is equal to the number of breaks in a minimum common partition of A and B.

The formula φ consists of $n-1$ clauses, with a clause C_i for each specific duo $a_i a_{i+1}, i \in [n-1]$. For $i \in [n-1]$, let $s_i = 1$ if a_i is the first occurrence of the letter a_i in A (that is, the other copy of the same letter occurs on a position $i' > i$), and let $s_i = 2$ otherwise (that is, if a_i is the second occurrence of the letter a_i in A). Similarly, let $t_i = 1$ if b_i is the first occurrence of the letter b_i in B and let $t_i = 2$ otherwise. We are ready to define φ. There will be three types of clauses in φ.

If the duo $a_i a_{i+1}$ does not appear in B at all, we define $C_i = 1$. The meaning is that in this case, $i, i+1$ is a break in A in any common partition of A and B. We call such a position an *inherent break*. Let b be the number of clauses of this type.

If the duo $a_i a_{i+1}$ appears once in B, say as $b_j b_{j+1}$, let $Y = X_{a_i}$ if $s_i \neq t_j$, and let $Y = \neg X_{a_i}$ otherwise; similarly, let $Z = X_{a_{i+1}}$ if $s_{i+1} \neq t_{j+1}$ and let $Z = \neg X_{a_{i+1}}$ otherwise. We define $C_i = Y \vee Z$. In this way, the clause C_i is satisfied if and only if $i, i+1$ is a break in a common partition consistent with the truth assignment of X_{a_i} and $X_{a_{i+1}}$.

Similarly, if the duo $a_i a_{i+1}$ appears twice in B, we set $C_i = X_{a_i} \oplus X_{a_{i+1}}$ if $s_i = s_{i+1}$, and we set $C_i = \neg X_{a_i} \oplus X_{a_{i+1}}$ otherwise, where $\oplus$ denotes the exclusive disjunction. Again, the clause C_i is satisfied if and only if $i, i+1$ is a break in a common partition consistent with

the truth assignment of X_{a_i} and $X_{a_{i+1}}$. Let k denote the number of these clauses.

By the construction, a truth assignment that satisfies the minimum number of clauses in $\varphi = C_1 \wedge \ldots \wedge C_{n-1}$ corresponds to a minimum common partition of A and B. In particular, the number of satisfied clauses is equal to the number of breaks in the common partition which is by one smaller than the number of blocks in the partition.

The formula φ resembles an instance of 2-SAT. However, 2-SAT formulas do not allow XOR clauses. One way to get around this is to replace every XOR clause by two OR clauses. This increases the length of the formula which in turn increases the resulting approximation ratio for 2-MCSP. In the rest of the section, we describe how to avoid this drawback.

Consider a duo $a_i a_{i+1}$ in A for which C_i is a XOR-clause. Then the duo $a_i a_{i+1}$ appears twice in B, and, by our assumption (1), the other occurrence of the letter a_i in A is followed by a letter different from a_{i+1} or the other occurrence of the letter a_i is the last letter in A. This implies that $k \leq b + 1$.

Let $\bar{\varphi}$ be the boolean formula derived from φ by omitting clauses of the first type, that is, $\bar{\varphi} = \bigwedge_{i:C_i \neq 1} C_i$. Let φ' be the formula that we get from $\bar{\varphi}$ by replacing each XOR clause $(X \oplus Y)$ by $(X \vee Y) \wedge (\bar{X} \vee \bar{Y})$ and keeping all other clauses. Since for any values of boolean variables X and Y, $(X \oplus Y) + 1 = (X \vee Y) + (\bar{X} \vee \bar{Y})$ (when using the boolean values of the parentheses as integers), the minimum number of satisfied clauses in $\bar{\varphi}$ is exactly by k smaller than the minimum number of satisfied clauses in φ'.

Let s be the minimum number of satisfied clauses in the formula $\bar{\varphi}$. Then, $s + b + 1$ is the size of a minimum common partition of A and B and the minimum number of satisfied

clauses in the 2-SAT formula φ' is $s + k$. An α-approximation for MIN 2-SAT instance φ' satisfies at most $\alpha \cdot (s+k)$ clauses and the same truth assignment satisfies at most $\alpha \cdot (s+k) - k$ clauses in $\bar{\varphi}$. Considering the additional b breaks for clauses of the first type, this truth assignment corresponds to a common partition with at most $\alpha \cdot (s + k) - k + b \leq \alpha \cdot (s + b + 1) - 1$ breaks. Since the size of the minimum common partition is $s + b + 1$, this is an α-approximation. For unsigned MCSP, the proof is completed.

For signed MCSP, we use the same correspondence between truth assignments and common partitions; the only difference is in definition of the clauses C_i. □

Chapter 5

Conclusion

In this dissertation, we developed computational methods for the identification and analysis of repetitive patterns in biological sequences. We gave a new definition of repeat that considers both length and frequency; we designed efficient algorithms for finding unique and popular oligos from large unigene databases; and we inferred the relation of multigene families between two genomes by studying the minimum common string partition (MCSP) problem.

Although our contributions are somewhat diverse, we can delineate some common strategies, which are detailed below.

Decompose long strings into short substrings. The rationale of this approach is that if a long string is repetitive, then some of its substrings will be repetitive as well. Shorter substrings serve the function of "signatures" for longer strings. The advantage of using shorter strings is that they are easier to store and process. An example of using this strat-

egy is the q-gram filtration algorithm for approximate string matching (see e.g. [55, page 162]). The filtration strategy is based on the observation that if two l-length strings A and B match with at most d mismatches, then they share an q-mer for $q = \lfloor \frac{l}{d+1} \rfloor$, i.e. $A[i \dots i+q-1] = B[i \dots i+q-1]$ for some $1 \leq i \leq l-q+1$. This strategy has been used in Chapter 2, where we define composite repeats as concatenations of elementary repeats, and in Chapter 3, where the algorithms for unique oligos and popular oligos use seeds and cores to prune the search space.

Organize strings with graphs. In order to extract structures from strings, it is often fruitful to organize them and their relationships using graphs (especially trees). The graph representation helps us isolate structures in the strings that otherwise will be difficult to detect. Graphs can also be implemented in space-efficient data structures for processing of strings, as shown in the various applications of suffix trees (see [34]). For instance, in Chapter 2 we use suffix trees to find exact elementary repeats in linear time; in Chapter 3 we use UPGMA trees to find the intersections of neighborhoods of candidate popular oligos; and in Chapter 4, we use conflict graphs to reduce MCSP problem to vertex cover problem.

Formulate combinatorial optimization problems. As it turns out, many problems in computational biology can be formulated as combinatorial optimization problems (see e.g. [33] and references therein). This strategy is based on the assumption that nature often favors parsimony models. For example, in Chapter 4 we look for an evolutionary scenario that minimizes the breakpoint distance between two genomes. Many of these problems are NP-hard

and one thus need to design approximation algorithms for them. However, we should use this strategy carefully because its success depends on whether the specific parsimony assumption is biologically realistic.

5.1 Future directions

In the problem of finding repeats in DNA sequences, the discovery of composite repeats is only sketched. Also, several questions on finding elementary repeats are still open. For example, it is not clear how to find consensus sequences of approximate repeats, how to rank the repeats according to their statistical significance, or how to decide the minimum length of nontrivial repeats, etc. We believe that the accuracy of determining elementary repeats is crucial in the identification of composite repeats.

In the future, we plan to combine the identification of composite and elementary repeats in a single process of dictionary and grammatical inference as follows. First, we will discover elementary repeats using a stochastic dictionary model, which has already been successfully applied to motif discovery (see e.g. [14]). Then, we will identify composite repeats using stochastic grammars, which have been successful in predicting RNA secondary structures (see [23] and references therein). The rationale of this approach lies in the observations that (i) motifs are one type of elementary repeats, and (ii) the structures in composite repeats, such as LTRs (Long Terminal Repeats), inverted repeats, etc., are somewhat related to the secondary structures of RNA.

A potential improvement for oligo design is to rank oligos according to their "usefulness", e.g. specificity for unique oligos and hybridization stability for popular oligos. A desirable feature for OLIGOSPAWN would consist of a ranking system that assigns a score to each candidate oligo.

Our approximation algorithms described in Chapter 4 are for the variants of MCSP that upper-bound the sizes of gene families. It is still unknown whether there exists a constant ratio bound for the general MCSP problem. For biological applications we need more flexible variants of MCSP. For example, if we use the more realistic modeling of genome rearrangement where two input genomes may contain different number of genes, then we should introduce additional operations such as gene duplications, insertions and deletions.

Bibliography

[1] Mark D. Adams, Jenny M. Kelley, Jeannine D. Gocayne, Mark Dubnick, Mihael H. Polymeropoulos, Hong Xiao, Carl R. Merril, Andrew Wu, Bjorn Olde, Ruben F. Moreno, Anthony R. Kerlavage, W. Richard McCombie, and J. Craig Venter. Complementary DNA sequencing: Expressed sequence tags and human genome project. *Science*, 252(5013):1651–1656, 1991.

[2] A. Apostolico, M. E. Bock, and S. Lonardi. Monotony of surprise and large-scale quest for unusual words (extended abstract). In G. Myers, S. Hannenhalli, S. Istrail, P. Pevzner, and M. Waterman, editors, *Proc. of Research in Computational Molecular Biology (RECOMB)*, pages 22–31, Washington, DC, April 2002.

[3] Adi Avidor and Uri Zwick. Approximating MIN k-SAT. In *Proceedings of 13th International Symposium on Algorithms and Computation (ISAAC)*, volume 2518 of *Lecture Notes in Computer Science*, pages 465–475, 2002.

[4] Timothy L. Bailey and Charles Elkan. Unsupervised learning of multiple motifs in biopolymers using expectation maximization. *Machine Learning*, 21(1/2):51–80, 1995.

[5] P. Baldi and S. Brunak. *Bioinformatics: The Machine Learning Approach. 2nd Ed.* The MIT Press, 2001.

[6] Zhirong Bao and Sean R. Eddy. Automated *De Novo* identification of repeat sequence families in sequenced genomes. *Genome Research*, 12(8):1269–1276, 2002.

[7] A Barakat, N Carels, and G Bernardi. The distribution of genes in the genomes of *Gramineae*. *Proc. Natl. Acad. Sci. U.S.A.*, 94:6857–6861, 1997.

[8] G. Benson. An algorithm for finding tandem repeats of unspecified pattern size. In S. Istrail, P. Pevzner, and M. Waterman, editors, *Proceedings of the 2nd Annual International Conference on Computational Molecular Biology*, pages 20–29, New York, NY, 1998. ACM Press.

[9] G. Benson. Tandem repeats finder – a program to analyze dna sequences. *Nucleic Acids Res.*, 27:573–580, 1999.

[10] Piotr Berman and Marek Karpinski. On some tighter inapproximability results. In *Proceedings of the of 26th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1644 of *Lecture Notes in Computer Science*, pages 200–209, 1999.

[11] M.S. Boguski, T.M. Lowe, and C.M. Tolstoshev. dbEST–database for "expressed sequence tags". *Nat. Genet.*, 4(4):332–3, 1993.

[12] E. T. Bolton and B. J. McCarthy. A general method for the isolation of RNA complementary to DNA. *Proc. Natl. Acad. Sci. U.S.A.*, 48(8):1390–1397, 1962.

[13] Jeremy Buhler and Martin Tompa. Finding motifs using random projections. *J. Comput. Bio.*, 9(2):225–242, 2002.

[14] Harmen J. Bussemaker, Hao Li, and Eric D. Siggia. Building a dictionary for genomes: Identification of presumptive regulatory sites by statistical analysis. *Proc. Natl. Acad. Sci. U.S.A.*, 97:10096–10100, 2000.

[15] A. Caprara. Sorting by reversals is difficult. In *Proceedings of the 1st Annual International Conference on Computational Molecular Biology*, pages 75–83, Santa Fe, NM, 1997. ACM Press.

[16] X. Chen, J. Zheng, Z. Fu, P. Nan, Y. Zhong, S. Lonardi, and T. Jiang. Assignment of orthologous genes via genome rearrangement. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 2(4):302–315, 2005.

[17] David A. Christie and Robert W. Irving. Sorting strings by reversals and by transpositions. *SIAM Journal on Discrete Mathematics*, 14(2):193–206, 2001.

[18] M. Chrobak, D. Eppstein, G.F. Italiano, and M. Yung. Efficient sequential and parallel algorithms for computing recovery points in trees and paths. In *ACM-SIAM Annual Symposium on Discrete Algorithms*, pages 158–167, 1991.

[19] Marek Chrobak, Petr Kolman, and Jiří Sgall. The greedy algorithm for the minimum common string partition problem. In *Proceedings of the 7th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX)*, pages 84–95, 2004. To appear in ACM Transactions on Algorithms.

[20] Won-Hyong Chung, Sung-Keun Rhee, Xiu-Feng Wan, Jin-Woo Bae, Zhe-Xue Quan, and Yong-Ha Park. Design of long oligonucleotide probes for functional gene detection in a microbial community. *Bioinformatics*, 21(22):4092–4100, 2005.

[21] T.J. Close, R. Wing, A. Kleinhofs, and R. Wise. Genetically and physically anchored EST resources for barley genomics. *Barley Genetics Newsletter*, 31:29–30, 2001.

[22] Graham Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. In *Proceedings of the 13th Annual ACM-SIAM Symposium On Discrete Mathematics (SODA)*, pages 667–676, 2002.

[23] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge University Press, 1998.

[24] R. Edgar and E. Myers. Piler: identification and classification of genomic repeats. In *Proc. of the 13th International Conference on Intelligent Systems for Molecular Biology (ISMB'05)*, page To appear, Detroit, Michigan, 2005. AAAI press, Menlo Park, CA.

[25] D. Eppstein. Personal communication, 2005. About the sparsest subsequence problem for selecting top unique oligo from unigenes.

[26] Eleazar Eskin and Pavel A. Pevzner. Finding composite regulatory patterns in DNA sequences. In *Proc. of the International Conference on Intelligent Systems for Molecular Biology*, pages Bioinformatics S181–S188. AAAI press, Menlo Park, CA, 2002.

[27] W. J. Ewens and G. R. Grant. *Statistical Methods in Bioinformatics. 2nd Ed.* Springer-Verlag, 2004.

[28] R. A. Fisher and F. Yates. *Example 12, Statistical tables*. London, 1938.

[29] Esra Galun. *Transposable elements: a guide to the perplexed and the novice with appendices on RNAi, chromatin remodeling and gene tagging*. Kluwer academic, 2003.

[30] M. R. Garey and D. S. Johnson. *Computers and intractability: a guide to the theory of NP-completeness*. Freeman, New York, NY, 1979.

[31] Avraham Goldstein, Petr Kolman, and Jie Zheng. Minimum Common String Partition Problem: Hardness and Approximations. In *Proceedings of the 15th International Symposium on Algorithms and Computation (ISAAC)*, volume 3341 of *Lecture Notes in Computer Science*, pages 484–495, 2004.

[32] Avraham Goldstein, Petr Kolman, and Jie Zheng. Minimum Common String Partition Problem: Hardness and Approximations. *Electronic Journal of Combinatorics*, 12(1), 2005.

[33] H.J. Greenberg, W.E. Hart, and G. Lancia. Opportunities for combinatorial optimization in computational biology. *INFORMS J. Comput.*, 16(3):211–231, 2004.

[34] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[35] Magnús M. Halldórsson. Approximating discrete collections via local improvements. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 160–169, San Francisco, California, 22–24 January 1995.

[36] C. S. Han, R. D. Sutherland, P. B. Jewett, M. L. Campbell, L. J. Meincke, J. G. Tesmer, M. O. Mundt, J. J. Fawcett. U-J. Kim, L. L. Deaven, and N. A. Doggett. Construction of a BAC contig map of chromosome 16q by two-dimensional overgo hybridization. *Genome research*, 10:714–721, 2000.

[37] CS Han, RD Sutherland, PB Jewett, ML Campbell, LJ Meincke, JG Tesmer, MO Mundt, JJ Fawcett, UJ Kim, LL Deaven, and NA Doggett. Construction of a BAC contig map of chromosome 16q by two-dimensional overgo hybridization. *Genome Research*, 104:714–721, 2000.

[38] J. M. Hancock and J. S. Armstrong. SIMPLE34: an improved and enhanced implementation for VAX and Sun computers of the SIMPLEx algorithm for analysis of clustered repetitive motifs in nucleotide sequences. *Comput. Appl. Biosci.*, 10:67–70, 1994.

[39] Sridhar Hannenhalli and Pavel A. Pevzner. Transforming cabbage into turnip: Polynomial algorithm for sorting signed permutations by reversals. *J. Assoc. Comput. Mach.*, 46(1):1–27, January 1999.

[40] G. Z. Hertz and G. D. Stormo. Identifying DNA and protein patterns with statistically sign ificant alignments of multiple sequences. *Bioinformatics*, 15:563–577, 1999.

[41] X. Huang and A. Madan. CAP3: A DNA sequence assembly program. *Genome Research*, 9:868–877, 1999.

[42] David S. Johnson. Approximation algorithms for combinatorial problems. *J. Comput. Syst. Sci.*, 9:256–278, 1974.

[43] I. Jonassen. Efficient discovery of conserved patterns using a pattern graph. *Comput. Appl. Biosci.*, 13:509–522, 1997.

[44] I. Jonassen, J. F. Collins, and D. G. Higgins. Finding flexible patterns in unaligned protein sequences. *Protein Science*, 4:1587–1595, 1995.

[45] Keich and Pevzner. Finding motifs in the twilight zone. In *Annual International Conference on Computational Molecular Biology*, pages 195–204, Washington, DC, April 2002.

[46] P. Kolman and T. Walen. Reversal distance for strings with duplicates: Linear time approximation using hitting set. Technical Report KAM-DIMATIA Series 776, Charles University in Prague, 2006.

[47] Petr Kolman. Approximating reversal distance for strings with bounded number of duplicates. In *Proceedings of the 30th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 3618 of *Lecture Notes in Computer Science*, pages 580–590, 2005.

[48] Stefan Kurtz, Jomuna V. Choudhuri, Enno Ohlebusch, Chris Schleiermacher, Jens Stoye, and Robert Giegerich. REPuter: The manifold applications of repeat analysis on a genomic scale. *Nucleic Acids Res.*, 29(22):4633–4642, 2001.

[49] Stefan Kurtz and Chris Schleiermacher. REPuter: Fast computation of maximal repeats in complete genomes. *Bioinformatics*, 15(5):426–427, 1999.

[50] C. E. Lawrence, S. F. Altschul, M. S. Boguski, J. S. Liu, A. F. Neuwald, and J. C. Wootton. Detecting subtle sequence signals: A Gibbs sampling strategy for multiple alignment. *Science*, 262:208–214, October 1993.

[51] Benjamin Lewin, editor. *Genes VIII*. Oxford University Press, New York, NY, 2004.

[52] Fugen Li and Gary D. Stormo. Selection of optimal DNA oligos for gene expression arrays. *Bionformatics*, 17(11):1067–1076, 2001.

[53] Dave Matthews, Thomas Wicker, and Jorge Dubcovsky. TREP: the triticeae repeat sequence database. Available at `http://wheat.pw.usda.gov/ITMI/Repeats/`.

[54] W Michalek, W Weschke, KP Pleissner, and A Graner. Est analysis in barley defines a unigene set comprising 4,000 genes. *Theor Appl Genet*, 104:97–103, 2002.

[55] G. Navarro and M. Raffinot. *Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.

[56] A.F. Neuwald, J.S. Liu, and C.E. Lawrence. Gibbs motif sampling: Detecting bacterial outer membrane protein repeats. *Protein Science*, 4:1618–1632, 1995.

[57] S. Ouyang and C.R. Buell. The TIGR plant repeat databases: a collective resource for the identification of repetitive sequences in plants. *Nucleic Acids Research*, 32:360–363, 2004.

[58] Pavel A. Pevzner and Sing-Hoi Sze. Combinatorial approaches to finding subtle signals in DNA sequences. In *Proc. of the International Conference on Intelligent Systems for Molecular Biology*, pages 269–278. AAAI press, Menlo Park, CA, 2000.

[59] Pavel A. Pevzner, Haixu Tang, and Glenn Tesler. *De novo* repeat classification and fragment assembly. In *Proc. of Research in Computational Molecular Biology (RECOMB)*, pages 213–222, San Diego, Ca, April 2004.

[60] A. L. Price, N. C. Jones, and P. A. Pevzner. De novo identification of repeat families in large genomes. In *Proc. of the 13th International Conference on Intelligent Systems for Molecular Biology (ISMB'05)*, page To appear, Detroit, Michigan, 2005. AAAI press, Menlo Park, CA.

[61] Sven Rahmann. Rapid large-scale oligonucleotide selection for microarrays. In *Proceedings of the First IEEE Computer Society Bioinformatics Conference (CSB'02)*, pages 54–63. IEEE Press, 2002.

[62] Isidore Rigoutsos and Aris Floratos. Combinatorial pattern discovery in biological sequences: The TEIRESIAS algorithm. *Bioinformatics*, 14(1):55–67, 1998.

[63] M. T. Ross, S. LaBrie, J. McPherson, and V. P. Stanton. Screening large-insert libraries by hybridization. In N.C. Dracopoli, J.L. Haines, B.R. Korf, D.T. Moir, C.C. Morton, C.E. Seidman, J.G. Seidman, and D.R. Smith, editors, *Current protocols in Human Genetics*, pages 5.6.1 – 5.6.52. John Wiley and Sons, New York, 1999.

[64] J.-M. Rouillard, C. J. Herbert, and M. Zuker. Oligoarray: Genome-scale oligonucleotide design for microarrays. *Bioinformatics*, 18(3):486–487, 2002.

[65] Steve Rozen and Helen J. Skaletsky. Primer3 on the WWW for general users and for biologist programmers. In S. Krawetz and S. Misener, editors, *Bioinformatics Methods and Protocols: Methods in Molecular Biology*, pages 365–386. Humana Press, Totowa, NJ, 2000. Available at `http://www-genome.wi.mit.edu/genome_software/other/primer3.html`.

[66] M. Sagot and E. W. Myers. Identifying satellites in nucleic acid sequences. In S. Istrail, P. Pevzner, and M. Waterman, editors, *Proceedings of the 2nd Annual International Conference on Computational Molecular Biology*, pages 234–242, New York, NY, 1998. ACM Press.

[67] D. Sankoff and N. El-Mabrouk. Genome rearrangement. In Tao Jiang, Ying Xu, and Michael Q. Zhang, editors, *Current Topics in Computational Molecular Biology*, pages 135–155. The MIT Press, 2002.

[68] J. P. Schmidt. All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings. In *Proceedings of the 3rd Israel Symposium on Theory of Computing and Systems*, pages 67–77. IEEE Computer Society Press, 1995.

[69] Dana Shapira and James A. Storer. Edit distance with move operations. In *13th Symposium on Combinatorial Pattern Matching (CPM)*, volume 2373 of *Lecture Notes in Computer Science*, pages 85–98, 2002.

[70] A.F.A. Smit and P. Green. REPEATMASKER. Available at `http://www.repeatmasker.org/`.

[71] D.L. Swofford. *PAUP: Phylogenetic Analysis Using Parsimony version 4.0 beta 10*. Sinauer Associates, Sunderland, Massachusetts, 2002.

[72] Martin Tompa and Jeremy Buhler. Finding motifs using random projections. In *Annual International Conference on Computational Molecular Biology*, pages 67–74, Montreal, Canada, April 2001.

[73] P. E. Warburton, J. Giordano, F. Cheung, Y. Gelfand, and G. Benson. Inverted repeat structure of the human genome: the x-chromosome contains a preponderance of large, highly homologoous inverted repeats that contain testes genes. *Genome Research*, 14:1861–1869, 2004.

[74] M. S. Waterman. *Introduction to Computational Biology*. Chapman & Hall, 1995.

[75] G. A. Watterson, W. J. Ewens, T. E. Hall, and A. Morgan. The chromosome inversion problem. *Journal of Theoretical Biology*, 99:1–7, 1982.

[76] Y. Yu, J. P. Tomkins, R. Waugh, D. A. Frisch, D. Kudrna, A. Kleinhofs, R. S. Brueggeman, G. J. Muehlbauer, R. P. Wise, and R. A. Wing. A bacterial artificial chromosome library for barley (*Hordeum vulgare L.*) and the identification of clones containing putative resistance genes. *Theor Appl Genet*, 101(7):1093–1099, 2000.

[77] J. Zheng, T. Close, T. Jiang, and S. Lonardi. Efficient selection of unique and popular oligos for large EST databases. In *Proceedings of Symposium on Combinatorial Pattern Matching (CPM'03)*, volume 2676 of *LNCS*, pages 273–283, Morelia, Mexico, June 2003. Springer.

[78] J. Zheng, T. Close, T. Jiang, and S. Lonardi. Efficient selection of unique and popular oligos for large EST databases. *Bioinformatics*, 20(13):2101–2112, 2004.

[79] J. Zheng and S. Lonardi. Discovery of repetitive patterns in dna with accurate boundaries. In *Proceedings of IEEE International Symposium on BioInformatics and BioEngineering (BIBE'05)*, pages 105–112, Minneapolis, Minnesota, October 2005.

[80] J. Zheng, J. T. Svensson, K. Madishetty, T. Close, T. Jiang, and S. Lonardi. Oligospawn: a software tool for the design of overgo probes from large unigene datasets. *BMC Bioinformatics*, 7(7), 2006.